

On Modular and Fully-Abstract Compilation – Technical Appendix

Marco Patrignani¹ Dominique Devriese²
Frank Piessens²

¹ Max Planck Institute for Software Systems, Germany

² iMinds-Distrinet, Dept. Computer Science, KU Leuven, Belgium
`{ first.last }@{ mpi-sws.org | cs.kuleuven.be }`

April 19, 2016

Abstract

Secure compilation studies compilers that generate target-level components that are as secure as their source-level counterparts. *Full abstraction* is the most widely-proven property when defining a secure compiler.

A compiler is *modular* if it allows different components to be compiled independently and then to be linked together to form a whole program.

Unfortunately, many existing fully-abstract compilers to untyped machine code are not modular. So, while fully-abstractly compiled components are secure from malicious attackers, if they are linked against each other the resulting component may become vulnerable to attacks.

This paper studies how to devise modular, fully-abstract compilers. It first analyses the attacks arising when compiled programs are linked together, identifying security threats that are due to linking. Then, it defines a compiler from an object-based language with method calls and dynamic memory allocation to untyped assembly language extended with a memory isolation mechanism. The paper provides a proof sketch that the defined compiler is fully-abstract and modular, so its output can be linked together without introducing security violations.

This paper uses colours to distinguish elements of different languages; please print this in colour.

1 Introduction

A compiler is a tool that, among other things, translates programs written in a source language into programs written in a target language. A compiler is *secure* when it preserves all security properties of the components (i.e., partial programs) it inputs in the components it outputs. Secure compilation studies compilers that generate target-level components that are as secure as their

source-level counterparts. Full abstraction is the most widely-adopted property to prove when defining a secure compiler [1, 4, 24, 35, 6, 7, 20, 2, 14, 18, 13]. A fully-abstract compiler preserves and reflects observational equivalence between the source components it inputs and the target components it produces. Observational equivalence captures security properties such confidentiality, integrity, etc, and since a fully-abstract compiler preserves all of them, it is a secure compiler.

A compiler is *modular* when it operates on components and its output can be *linked* together into larger components (and possibly into whole programs).

With the advent of machine-code level security architectures (e.g., protected modules architectures (*PMA*) [39, 31, 30], capability machines [43, 42], micro-policies enforcing architectures [9], address space layout randomisation [4], etc.) researchers have investigated secure (fully-abstract) compilation to such architectures [35, 26, 4, 24].

For secure compilation to *PMA* (informally, a low-level memory isolation mechanism), only compilation of a single protected module has been considered. Generalizing this to multiple modules, supporting modular compilation and linking of protected modules at machine code level, would be useful for a number of reasons: (i) code is easier to develop and distribute in standalone components and (ii) merging modules in a single one is not always possible nor desirable. Concerning (i), standard arguments for separate compilation apply. Particularly, it is more efficient to compile only the modules that changed and link it against the previously compiled other code as opposed to recompiling the entire code base. Moreover, it is common in practice and useful to use libraries that are built from safe source code but only available in compiled form. When targeting *PMA* with attestation capabilities, a benefit of having each module in a different protected module is that this makes it possible to attest them independently. Concerning (ii), there can be several reasons why merging more modules in a single one is not desirable; even with a secure compiler, it can still be useful to protect securely compiled components from each other. Consider a source language that allows programmers to write both safe and unsafe code at the same time (e.g., Rust, C# unsafe blocks or the language considered by Juglaret *et al.* [27]). A desirable property for a compiler for such a language is that unsafe code does not affect the safe one. In this case, one can foresee splitting the safe and the unsafe program parts in two different modules and using the secure compiler presented in this paper for the safe part. In this way, both the safe and the unsafe code cannot be directly tampered with by an attacker and the unsafe code can not affect the safe one. Another setting where merging multiple modules is undesirable is when a programmer provides a hand-optimised securely-compiled component that performs better than the ones obtained by a secure compiler, but with the same security guarantees. In this case, merging such a component with other ones in a single module seems undesirable, as the hand-made assembly can be error-prone and maliciously-crafted. However, by using a modular secure compiler, other components can still interact with the hand-optimised one and no one's security can be tampered.

Extending the compiler to support modular compilation is, however, sur-

prisingly complicated for a number of reasons:

1. with a single protected module, all run-time meta information about the execution that needs to be protected (e.g., the call stack, or dynamic type information) can be stored within that single protected module. For a modular compiler, that state must be divided over the various protected modules, or – possibly – stored in a centralized trusted protected module.
2. with a single protected module, object references are either private to the module or public. With support for multiple protected modules, object references can also be shared between some modules and still be unknown to other ones. This requires a mechanism for protecting object references that is more elaborate than what the existing secure compiler to *PMA* uses.

There are several interesting approaches to address these challenges. This paper investigates how these issues can be addressed constructing a modular fully-abstract compiler for *PMA* in the style of Intel SGX [30] *without* additional hardware support. Even if this turns out to be intricate, we believe there is value in this approach, as this kind of hardware support is available on commercially available systems today, whereas more advanced hardware support is only available in research prototypes.

An alternative approach to modular, secure compilation would be to investigate what kind of novel hardware security architectures can help in addressing these challenges. This is the approach taken by Juglaret *et al.* [26] and it can likely lead to a secure compilation scheme. An important downside is that it may take a while before such hardware extensions are available in mainstream systems.

The main contribution of this paper is $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$, a modular and fully-abstract compiler to SGX-like *PMA*. The source language of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is *JEM*, an object-based imperative language with method calls and dynamic memory allocation. The target language of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is *AIL*, untyped assembly language with an explicit linking mechanism extended with *PMA*. *JEM*, *AIL* and *PMA* are formalised in Section 2. Explicitly considering linking between assembly components generates a number of previously-unobserved problems that are presented in Section 3. The main contribution of this paper is the formalisation of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$, presented in Section 4. Supporting additional language features is discussed in Section 5. The proof sketch of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ being fully-abstract and modular is presented in Section 6. Finally, Section 7 discusses related work and Section 8 concludes.

2 Languages Formalisation

This section describes *JEM* (Section 2.1) and *AIL* (Section 2.2), respectively the source and the target language of the secure compiler $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$.

2.1 The Source Language JEM

JEM is a strongly-typed, single-threaded, object-based imperative language that has **private** fields and **public** methods; it does not allow any undefined behaviour to arise. JEM is presented in a *green* font (Figure 1).

<i>component</i>	$\mathcal{C} ::= \overline{\mathcal{C}}$
<i>classes</i>	$C ::= \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O}$
<i>objects</i>	$O ::= \text{object } o : t\{\overline{F}\}$
<i>class declarations</i>	$I ::= \text{class-decl } c\{\overline{m} : \overline{M}_t\}$
<i>object declarations</i>	$X ::= \text{obj-decl } o : t;$
<i>methods</i>	$M ::= \text{public } m(\overline{x}) : M_t \{\text{return } E;\}$
<i>signatures</i>	$M_t ::= t(\overline{t}) \rightarrow t$
<i>fields</i>	$F ::= \text{private } f = v$
<i>field types</i>	$F_t ::= f : t$
<i>constructors</i>	$K ::= c(\overline{f} : \overline{t}) \{\text{this}.\overline{f}' = \overline{f}\}$
<i>types</i>	$t ::= \text{Unit} \mid \text{Bool} \mid \text{Int} \mid c \mid \text{Obj}$
<i>values</i>	$v ::= \text{unit} \mid \text{true} \mid \text{false} \mid n \mid o$
<i>operations</i>	$\text{op} ::= + \mid - \mid == \mid \dots$
<i>expressions</i>	$ \begin{aligned} E ::= & v \mid x \mid E.f \mid E.f = E \mid E.m(\overline{E}) \\ & \mid E \text{ op } E \mid \text{new } t(\overline{E}) \mid E; E \mid \text{this} \\ & \mid \text{if } (E) \{E\} \text{ else } \{E\} \mid \text{exit } E \\ & \mid \text{instanceof}(E : c) \mid \text{var } x : t = E \end{aligned} $

Figure 1: Syntax of JEM; lists of elements $a_1 \dots a_n$ are denoted as \overline{a} .

A class C declares (external) classes and objects it requires (these are called **import** requirements) then it defines its constructor, fields, methods and objects implementing that class. Objects of a class can only be allocated by methods of that class (so cross-component memory allocation happens via factory methods). Class declarations I define class signatures, i.e., the class name and the methods implemented by that class. Object declarations X are references to objects implementing a different class. A JEM component \mathcal{C} is a collection of classes $\overline{\mathcal{C}}$. If all **import** requirements of $\overline{\mathcal{C}}$ are satisfied by some other class in $\overline{\mathcal{C}}$, then $\overline{\mathcal{C}}$ is a whole program. Two components \mathcal{C}_1 and \mathcal{C}_2 satisfy each other, denoted with $\mathcal{C}_1 \cap \mathcal{C}_2$, if all **import** requirements of \mathcal{C}_1 are classes and objects in \mathcal{C}_2 and vice-versa.

The top of the JEM class hierarchy is **Obj**, a class defining no methods. All

classes implicitly extend `Obj`; JEM does not provide any other form of inheritance. Primitive types are `Unit`, inhabited by `unit`, `Bool`, inhabited by `true` and `false` and `Int`, inhabited by natural numbers n . Identifiers for classes c , objects o , methods m , fields f and variables x are taken from distinct denumerable sets.

The semantics of JEM is standard and unsurprising; it is omitted for space reasons.

The security mechanism of JEM is given by `private` fields, which can be used to define security properties such as confidentiality and integrity (as defined in Section 3.1).

2.1.1 Contextual Equivalence for JEM

To reason about the behaviour of JEM components, contextual equivalence is used [37]. Contextual equivalence is the coarsest relation that tells when two components are behaviourally equivalent; its definition (Definition 1) is rather standard [6, 7, 35, 20, 4, 24, 13, 21, 32].

Contexts \mathbb{C} are partial programs with a hole ($[\cdot]$), formally $\mathbb{C} ::= \overline{\mathbb{C}}[\cdot]$. The plugging of a component \mathcal{C} in a context \mathbb{C} , denoted as $\mathbb{C}[\mathcal{C}]$, returns a whole program $\mathbb{C};\mathcal{C}$. There are two (common) assumptions for the plugging to succeed. The first one is that $\mathcal{C} \cap \mathbb{C}$, the second one is that \mathcal{C} and \mathbb{C} are well-typed. If any assumption is not upheld, the plugging returns the empty program.

Definition 1 (Contextual equivalence for JEM). $\mathcal{C}_1 \simeq_{ctx} \mathcal{C}_2 \triangleq \forall \mathbb{C}, \mathbb{C}[\mathcal{C}_1] \uparrow \iff \mathbb{C}[\mathcal{C}_2] \uparrow$, where \uparrow means divergence, i.e., the execution of an unbounded number of reduction steps.

2.2 PMA and the Target Language AIL

AIL (acronym of Assembly plus Isolation and Linking) is a low-level language that models a von Neumann machine enhanced with *PMA* (Section 2.2.1), with an idealised form of cryptographic nonces (Section 2.2.2) and with an explicit linking mechanism (Section 2.2.3). AIL is the adaptation of an analogous language that had no linking mechanism, no idealised cryptographic nonces and a single *PMA* module [35, 36]. After presenting *PMA* and AIL, this section describes the semantics (Section 2.2.4) and contextual equivalence for AIL (Section 2.2.5).

2.2.1 PMA

PMA is an assembly-level isolation mechanism based on program counter-based memory access control. *PMA* can be implemented in software (e.g., via a hypervisor) [29, 38, 39] or in hardware [8, 30, 31]; Intel is bringing it to mainstream processors with the Intel SGX instruction set [30]. *PMA* logically divides the memory space into several protected and one unprotected section; a protected section is called *protected module*, each module has a unique *module id*. All protected sections are further divided into a *code* and a *data* section. Code sections contain a variable number of *entry points*: the only protected addresses to which instructions in unprotected memory or in other protected sections can

jump. Entry points allow code from within the module to interoperate with external code. Data sections are only accessible from within the protected code section of the same module. The table below summarises the *PMA* access control model.

From \ To	Unprotected	Protected		
		Entry Point	Code	Data
Unprotected	r w x	x		
Protected	r w x	Same id		
		r x	r x	r w
		Different id		
		x		

2.2.2 Core AIL

AIL is presented in a *pink* font (Figure 2).

<i>numbers</i>	$n ::= n \in \mathbb{N}$
<i>words</i>	$w ::= n \mid \sigma \mid \pi$
<i>addresses</i>	$a ::= (id, n)$
<i>memories</i>	$m ::= \overline{a \mapsto w}$
<i>flags</i>	$f ::= \overline{ZF \mapsto 0 \mid 1; SF \mapsto 0 \mid 1}$
<i>register files</i>	$r ::= \overline{\mathbf{r} \mapsto w}$
<i>module ids</i>	$id \in \mathcal{ID} \subset \mathbb{N}$
<i>symbolic nonces</i>	$\pi \in \Pi$
<i>module descriptors</i>	$s ::= (id, n_c, n)$
<i>nonce oracles</i>	$h ::= \overline{\pi}$
<i>symbols</i>	$\sigma, \iota \in \mathcal{S}$
<i>instructions</i>	$i \in \mathcal{I} \subset \mathbb{N}$

Figure 2: Formalisation of AIL (part 1: language).

Words w are either natural numbers n (including instruction encodings and module ids), symbols σ (explained in Section 2.2.3) or symbolic nonces π . Addresses a are pairs of natural numbers and module ids, so a module with id id has an infinite memory starting from address $(id, 0)$. Unprotected memory has id 0. The registers file r contains an unbounded number of registers \mathbf{r} . The flags register f contains a sign (SF) and a zero flag (ZF) that are set by arithmetic and testing instructions. Memories m are lists of bindings from addresses to words. Module descriptors s are triples that define a module memory layout: id indicates the id of the module, n_c is the length (in number of addresses) of the code section and n is the number of entry points. Between each entry point there are a fixed amount of addresses that are not entry points, indicate

this number of addresses with \mathcal{N}_w . The first entry point is located at address 0, the second one at address \mathcal{N}_w and so on; the last one is located at address $n \cdot \mathcal{N}_w$ such that $n \cdot \mathcal{N}_w < n_c$. Symbolic nonces π are non-guessable, unforgeable tokens; they model what a program could create using cryptographic primitives or unguessable randomisation [19, 40]. For the sake of simplicity, they are considered to be 0 for arithmetic operators.¹ Nonce oracles h are used as suppliers of fresh nonces for the **new** instruction (see Rule **Eval-new** in Figure 4). Instructions i are elements of the set \mathcal{I} (Table 1) that define the programming language executed on the architecture.

movl $r_d \ r_s \ r_i$	Load the word from the address in registers (r_s, r_i) into register r_d .
movs $r_d \ r_s \ r_i$	Store the contents of register r_s at the address found in registers (r_d, r_i) .
movi $r_d \ k$	Load the constant value k into register r_d .
add $r_d \ r_s$	Write $r_d + r_s$ into register r_d and set the ZF flag accordingly.
sub $r_d \ r_s$	Write $r_d - r_s$ into register r_d and set both the ZF and the SF flags accordingly.
cmp $r_s \ r_d$	Compare r_s and r_d and set the flags accordingly.
jmp $r_d \ r_i$	Jump to the address located in register r_d in the module with id r_i .
je $f_i \ r_i$	If flag f_i is set, jump to the address in register r_i in the current module.
zero	Set all registers to 0.
new r_d	Initialise register r_d with a fresh symbolic nonce.
halt	Stop the execution.

Table 1: Instruction set \mathcal{I} .

2.2.3 Modules, Programs and Linking (Figure 3)

To deal with linking, **AtL** has symbols σ, ι . Symbols in a module are placeholders for words that will be filled when the module is linked against another one; symbols can be found in memory and in the symbol table. A symbol table t contains exported method and object bindings as well as required method and object bindings. An exported method binding **EM** (resp. object binding **EO**) maps a method name m and its type M_t (resp. an object name o and its class c) to the address a where the method (resp. class) is located. A required method binding **RM** (resp. object binding **RO**) maps a method name m and its type M_t (resp. an object name o and its class c) to the id symbol ι and the number symbol σ used for it. A method (resp. object) export binding fulfils a method (resp. object) requirement if both bind the same method name and type (resp. object name and class).

¹ Having arithmetic operations affect nonces would only let us model guessing attacks on them, but they are assumed to be resilient to these attacks so we chose this option for simplicity.

<i>modules</i>	$M ::= (m; s; t)$	<i>programs</i>	$P ::= (m; \bar{s}; t)$
<i>symbol tables</i>		$t ::= \overline{EM}; \overline{EO}; \overline{RM}; \overline{RO}$	
<i>exported methods</i>		$EM ::= m : M_t \mapsto a$	
<i>exported objects</i>		$EO ::= o : c \mapsto n$	
<i>required methods</i>		$RM ::= m : M_t \mapsto \iota; \sigma$	
<i>required objects</i>		$RO ::= o : c \mapsto \sigma$	

Figure 3: Formalisation of A_{IL} (part 2: linking).

A single A_{IL} module M is a triple listing a memory m , its module descriptor s and a symbol table. An A_{IL} program P is a collection of modules whose module descriptors are listed in \bar{s} . If the required bindings of a program are empty lists, then the program is whole. When a program is not whole it is partial, i.e., it is a collection of modules with a symbol table with unfulfilled requirements. Informally, two A_{IL} programs P_1 and P_2 satisfy each other, denoted with $P_1 \frown P_2$, if all the required bindings of P_1 are fulfilled by a binding in P_2 and vice-versa. Modules and programs are well-formed if the only symbols in their memory are those captured in the symbol table. In the following, we only consider well-formed programs and modules.

Two memories (resp. two module descriptors) agree if their domains are disjoint (resp. if their ids are distinct). Two A_{IL} modules P_1 and P_2 can be joined together, denoted with $P_1 + P_2$, only if the two memories and if the memory descriptors agree. Joining of modules results in a program that is the concatenation of the memories, memory descriptors and symbol tables of P_1 and P_2 . Joining of symbol tables results in a new symbol table. If a requirement (method or object) is fulfilled by a requirement in another table, that requirement is removed from the resulting table. Moreover, in the resulting memory, all symbols of fulfilled requirements are replaced with what is bound by the fulfiller requirement. Consider P_1 to require method $m : M_t \mapsto \iota; \sigma$ and P_2 to export method $m : M_t \mapsto (id, n)$. When merging P_1 and P_2 , all occurrences of ι and σ in the memory of P_1 will be replaced with id and n , respectively. The resulting symbol table will no longer have the requirement for $m : M_t$ as that has been fulfilled.

2.2.4 Dynamic Semantics of A_{IL}

The dynamic semantics of whole programs (\rightarrow) relates program states (p, r, f, m, s, h) where p is the program counter (i.e., an address). This semantics relies on another one for modules (\xrightarrow{id}); the latter tells when a module with id id performs a reduction (Rule [Eval-single-to-multiple-modules](#)). Figure 4 shows an excerpt of the reduction rules. Memory access is indicated as: $m(a) = w$; memory update is indicated as $m[a \mapsto w]$. The same notation is adopted for

$$\begin{array}{c}
\text{(Eval-new)} \\
\frac{p = (id, n) \quad m(p) = \text{new } r_d \quad r' = r[r_d \mapsto \pi] \quad \pi \notin h}{(p, r, f, m, s, \pi; h) \xrightarrow{id} ((id, n+1), r', f, m, s, h)} \\
\text{(Eval-module)} \\
\frac{(p, r, f, m, s, h) \xrightarrow{id} (p', r', f', m', s', h') \quad p \vdash \text{currentModule}(\bar{s}, s) \quad p = (id, n)}{(p, r, f, m, \bar{s}, h) \rightarrow (p', r', f', m', \bar{s}, h')} \\
\text{(Eval-movs)} \\
\frac{p = (id, n) \quad m(p) = (\text{movs } r_d \ r_s \ r_i) \quad \bar{s} \vdash \text{writeAllowed}(n, (r(r_d), r(r_i))) \quad m' = m[(r(r_d), r(r_i)) \mapsto r(r_s)]}{(p, r, f, m, \bar{s}, h) \rightarrow ((id, n+1), r, f, m', \bar{s}, h)} \\
\text{(Eval-jmp)} \\
\frac{p = (id, n) \quad m(p) = (\text{jmp } r_d \ r_i) \quad n' = r(r_d) \quad id' = r(r_i) \quad r' = r'[r_0 \mapsto id] \quad \bar{s} \vdash \text{validJump}(p, (id', n'))}{(p, r, f, m, s, h) \rightarrow ((id', n'), r', f, m, s, h)}
\end{array}$$

Figure 4: An excerpt of the dynamic semantics of **AIL**

register files and flags register access and update. Assumptions of the form $\bar{s} \vdash \text{name}(a, a')$ model the enforcement of the *PMA* access control policy; their names are self-explanatory. For example, $s \vdash \text{writeAllowed}(n, n')$ tells if an address n' can be written from another address n and $\bar{s} \vdash \text{validJump}(a, a')$ tells if address a' is executable (x) from address a . $p \vdash \text{currentModule}(\bar{s}, s)$ tells whether the program counter p is in a specific module whose descriptor is s where s is within \bar{s} . Instruction **jmp** is the only instruction that can perform cross-module jumps (Rule **Eval-jmp**). When a cross-module jump is performed, the *PMA* architecture inserts the *id* of the caller in register r_0 . This functionality is called caller-callee authentication; some *PMA* implementations provide it [39, 31] and it is achievable on Intel SGX by means of the EREPORT instruction [16]. Whenever the execution of an instruction violates the *PMA* access control policy (e.g., jumping to a module address that is not an entry point), the execution is terminated.

2.2.5 Contextual Equivalence for **AIL**

Analogously to **JEM**, to reason about **AIL** components we define when they are contextually equivalent. Contexts \mathbb{P} are partial programs with a hole ($[\cdot]$) that can be filled with a component in order to create a whole program, formally $\mathbb{P} ::= P[\cdot]$. Plugging a component P in a context \mathbb{P} , denoted as $\mathbb{P}[P]$, returns a whole program $\mathbb{P}+P$, given that $P \cap \mathbb{P}$.

Contextual equivalence for languages with oracles relies on a notion of contextual preorder [3] which is needed to ensure obvious equivalences are satisfied (Example 1).

Example 1 (Need for preorders). Consider a module P_{n2} that generates two nonces and returns only one and another one P_{n1} that generates and returns

one nonce only. These programs are intuitively equivalent, but a contextual equivalence definition analogous to that for JEM simply does not yield this fact because we cannot quantify over all oracles and use the same oracle for both programs. Given that P_{n1} runs with an oracle h_1 , P_{n2} has the same behaviour if it runs with an oracle h_2 that has all elements of h_1 interleaved with fresh elements. Dually, given that P_{n2} runs with an oracle h_2 , P_{n2} has the same behaviour if it runs with an oracle h_1 that has only the even-numbered elements of h_2 . \square

Definition 2 (Contextual preorder for $\text{A}\mathbb{U}$). $P_1 \sqsubset P_2 \triangleq \forall \mathbb{P}, h_1, \exists h_2. \mathbb{P}[P_1], h_1 \uparrow \Rightarrow \mathbb{P}[P_2], h_2 \uparrow$, where $\mathbb{P}[P], h$ indicates the initial state of $\mathbb{P}+P$ with oracle h .

Definition 3 (Contextual equivalence for $\text{A}\mathbb{U}$). $P_1 \simeq_{ctx} P_2 \triangleq P_1 \sqsubset P_2$ and $P_2 \sqsubset P_1$.

3 Security Pitfalls

This section describes the threat model considered in this paper (Section 3.1) and the security pitfalls that arise when linking is explicitly considered (Section 3.2).

3.1 The Threat Model

The goal of this section is to familiarise the reader with the security aspects of secure compilation that are relevant for this paper. We do not provide an in-depth analysis of these aspects as they are not the focus of this paper, though we believe these insights can be helpful for many readers.

Secure compilation papers generally consider a threat model for an attacker with target-level code injection capabilities that operates, for example, by exploiting a bug in the system and injecting or loading arbitrary target programs. His goal is to violate the security properties of compiled programs found in the system. Specifically, in this paper we consider an attacker that can load arbitrary malicious code.

In this paper, the system is the von Neumann machine formalised by the $\text{A}\mathbb{U}$ language, where multiple protected modules are found. The compiled programs that the attacker wants to violate span *some* of the protected modules, as explained in Section 1; their security properties must not be violated. The attacker is modelled as code (and data) that spans unprotected memory as well as *other* protected modules; thus the attacker is modelled as an $\text{A}\mathbb{U}$ context. This attacker can interact with any of the compiled programs whose security is of interest in any way that the assembly language (and the *PMA* access control policy) allow him. Thus, the attacker must respect the processor-enforced *PMA* access control policy and, most importantly, he cannot tamper with it.

We are interested in program security properties that can be expressed by means of contextual equivalence; some examples include confidentiality and integrity. A value is confidential if it cannot be discerned by other components

than the one declaring it. In other words, a value v in a component \mathcal{C} is confidential if \mathcal{C} is contextually-equivalent to \mathcal{C}' which is \mathcal{C} with a different value for v . Integrity of a value means that it cannot be modified by other components than the one declaring it. In other words, a value v in a component \mathcal{C} has integrity if \mathcal{C} is contextually-equivalent to \mathcal{C}' which is \mathcal{C} where every interaction with other component is followed by a check that the value of v is the same as before the interaction.

The goal of the paper is to provide a compiler $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ that produces secure AIL modules. Thus, $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ takes JEM components that possibly has some security properties and outputs AIL modules that has the same security properties. If this holds, then the attacker's goal is nullified.

As compiler full-abstraction is preservation and reflection of contextual equivalence, proving $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ to be fully-abstract implies that $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is secure.

3.2 Security Problems Related to Linking

Problems 1 to 4 are full-abstraction violations that arise in the presence of linking. Any full-abstraction violation can be lead back to confidentiality or integrity violations.

Problem 1 (Object id guessing). *Consider an object allocated at address a in module M_1 and that is only shared between two modules M_1 and M_2 . As object ids are just addresses in memory, nothing prevents another module M_3 to guess address a and call methods on it. Note however, that this violates the intended confidentiality of a , that is supposed to be visible only to M_1 and M_2 .*

A naïve solution to this problem is tracking which module will receive a certain object id in order to detect and stop guesses from modules that have not received that object id. However, this does not scale to the case where M_2 forwards a to M_3 . Since the forwarding is done outside of M_1 , M_1 has no way of adding M_3 to the modules that are allowed to access a . As this may not be known statically, a different solution is needed.

To address this concern, target-level object ids must be unguessable. ■

Problem 2 (Call stack shortcutting). *As all assembly languages, AIL calls and returns are jumps to addresses in memory. Consider the following sequence of cross-module function calls: $M_2 \rightarrow M \rightarrow M_1 \rightarrow M'$, where primes are used to denote subsequent calls to the same module.*

M' can return to M_2 , as it learnt the address to return there when M_2 called it. However, that instruction bypasses the rest of the call stack and particularly M_1 in a way that is not possible in the source language, where control flow follows a well-bracketed sequence of calls/returns.

To address this concern, securely compiled code must ensure a well-bracketed traversal of the stack. ■

Problem 3 (Types of objects in other modules). *Consider an object of type t' allocated at address a in module M_1 and a method m compiled in a different module M_2 that takes a parameter of type t . t and t' are different. A third*

module M_3 could pass a as an argument to the compilation of m , violating the JEM assumption that only well-typed programs are executed. A dynamic check on the type of a should be made, but the code of m resides in M_2 and it has no way to access a , which resides in module M_1 .

To address this concern, dynamic typechecks must be made on all objects that are passed and received via methods and the encoding of the class of an object must be accessible outside of the module containing the object. ■

Problem 4 (Existence of objects in other modules). Analogously, to Problem 3, in place of an object of the wrong type, M_3 code can call the compilation of m by passing a non-existent object id. A non-existent object id is a non-**null** object id that does not point to an object, so that calling methods on it will fail.

To address this concern, when securely compiled code receives an object from another module, it must assess whether that object exists or not. ■

4 A Secure Compiler from JEM to AIL

$\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is a modular, two-step compiler that inputs a JEM class C and returns two AIL modules. Formally:

$$\llbracket C \rrbracket_{\text{AIL}}^{\text{JEM}} = \text{protect}(\llbracket C \rrbracket) + \llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}.$$

$\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ consists of a call to the $\llbracket \cdot \rrbracket$ function (Section 4.1), followed by a call to the $\text{protect}(\cdot)$ one (Section 4.3). The result is joined with an additional module $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$ (Section 4.2).

Intuitively, $\llbracket \cdot \rrbracket$ is a compiler responsible for correctly translating classes, methods and objects to assembly. $\text{protect}(\cdot)$ is a wrapper around the code generated by $\llbracket \cdot \rrbracket$ that prevents direct access to that code and ensures that any interaction with that code is regulated to behave as valid JEM code. This wrapping ensures that the output of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is secure. $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$ is a central, trusted module that operates as a monitor, regulating the structure of function calls and keeping information of allocated objects.

$\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is modular, i.e., it can be applied to JEM components C . In this case $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ recursively calls itself on all classes composing the JEM component. The secure linker $\text{link}(\cdot)$ combines all compiled components to a low-level component (Section 4.4). Formally:

$$\llbracket C_1; \dots; C_n \rrbracket_{\text{AIL}}^{\text{JEM}} = \text{link}(\llbracket C_1 \rrbracket_{\text{AIL}}^{\text{JEM}}, \dots, \llbracket C_n \rrbracket_{\text{AIL}}^{\text{JEM}})$$

The overall structure of the compiler proposed in this paper is similar to the one for a single-module version of PMA [35], but two important extensions are needed to handle the issues (Items 1 and 2) mentioned in Section 1:

1. we introduce a small centralized trusted module $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$ that keeps information about the global control flow of the program, and about types

of objects shared between multiple modules. A key challenge is to design this module such that the information it exposes does not break full abstraction (Section 4.2).

2. we introduce unforgeable object references. Since we do not want to assume hardware support for this (as is done for instance in capability machines), we solve this by assuming the existence of a secure random number generator that can create unguessable random numbers (nonces) that are sufficiently large to make brute-force attacks infeasible (as achievable via the `rand` instruction of intel processors [22]). We model these nonces symbolically (i.e., π) and use them to represent references to objects outside of the module where the object is defined (Section 4.3.1).

4.1 The First Step: $\langle \cdot \rangle$

$\langle \cdot \rangle$ is a compiler that translates a single JEM class to AIL code, data and a symbol table. Instead of giving a specific instance of $\langle \cdot \rangle$, we define what assumptions (Assumption 1, 2 and 4) such a compiler must uphold for the full-abstraction result to hold. The full-abstraction result is achieved parametrically for any $\langle \cdot \rangle$ that upholds those assumptions.

Assumption 1 (Output of $\langle \cdot \rangle$). *The compilation of a JEM class returns a memory m_c providing code implementing methods \overline{M} , a memory m_d providing data implementing objects \overline{O} and a symbol table t . t contains exported and required methods and objects bindings sorted lexicographically by class and then method or object name [35].*

The produced code and data implement a class-local stack that is used to perform calls and returns between methods.

$\langle \cdot \rangle$ is multi-entry i.e., it exports an address for each method that can be called. $\langle \cdot \rangle$ is single-exit i.e., all jumps outside the code are performed by a common piece of code located in a known part of the produced code: m_{exit} . $\langle \cdot \rangle$ expects the address where `instanceof` is implemented to be supplied later, so a call to `instanceof` is compiled to symbols $(\iota_{inst}, \sigma_{inst})$ which capture both parts of the address where `instanceof` is. This is captured by the required bindings with a binding of the form $instanceof : Obj(Obj, Obj)Bool \mapsto (\iota_{inst}, \sigma_{inst})$ which must be in t .

JEM object ids o are compiled to numbers n that point to a memory region with type information and fields.

Formally: $\langle import \overline{I}; \overline{X}; class c\{K \overline{F_t} \overline{M}\}; \overline{O} \rangle = (m_c + m_{exit} + m_{inst}; m_d; t)$.

We overload the $\langle \cdot \rangle$ notation and use it to indicate values as are compiled by $\langle \cdot \rangle$. Indicate the compilation of `unit` as $\langle unit \rangle$ and the compilation of `true` as $\langle true \rangle$. Analogously, indicate the encoding of JEM types t in AIL as $\langle t \rangle$. For example, indicate the encoding of `Unit` as $\langle Unit \rangle$ and the encoding of a class type c as a natural number $\langle c \rangle = n$.

Assumption 2 (Calling convention of $\langle \cdot \rangle$). *Registers are used according to the following calling convention. r_0 is used by caller-callee authentication to*

store the caller module id. \mathbf{r}_1 to \mathbf{r}_4 are used as general working registers, so they do not contain relevant information. \mathbf{r}_5 identifies the return address in a `jmp` that models a method call; if \mathbf{r}_5 is 1, then that `jmp` is interpreted as a return. \mathbf{r}_6 identifies the current object (`this`) in a target-level method call or the returned value in a target-level return. \mathbf{r}_7 onwards are used to communicate method parameters.

Assumption 3 (Restrictions of $\llbracket \cdot \rrbracket$). *Compiled components do not use the `exit E` expression. Additionally, they do not read nor write to unprotected memory.*

Assumption 4 (Correctness of $\llbracket \cdot \rrbracket$). *$\llbracket \cdot \rrbracket$ must be correct and adequate, i.e., it translates any JEM expressions and values into AIL code that behaves in the same way [23, 12, 28].*

4.2 The System Module: $\llbracket Sys \rrbracket_{AIL}^{JEM}$

The system module $\llbracket Sys \rrbracket_{AIL}^{JEM}$ is a central, trusted module that provides functionality used by all securely-compiled components. $\llbracket Sys \rrbracket_{AIL}^{JEM}$ contains: a global store \mathcal{G} (Section 4.2.1), a global call stack \mathcal{S} (Section 4.2.2) and functionality to interact with \mathcal{G} and \mathcal{S} (Section 4.2.3); its definition is provided last (Section 4.2.4).

In the following, when code *aborts* we mean that all registers and flags are reset, then `halt` is executed.

4.2.1 The Global Store \mathcal{G}

\mathcal{G} tracks globally-known object ids (i.e., ids that are not just local to a module), their type and the module id where the object resides. Formally $\mathcal{G} = \overline{w \mapsto w', id}$, where w is an object id and w' is a class encoding. Retrieval from \mathcal{G} is denoted with $\mathcal{G}(w)$ and addition to it is denoted with $\mathcal{G} + (w \mapsto w', id)$.

$\llbracket Sys \rrbracket_{AIL}^{JEM}$ provides two entry points where the following procedures are implemented: `testObj(w, w')` and `registerObj(w, w')`. The former tells if an object w exists and implements a certain class w' . The latter adds a new binding to \mathcal{G} , the new binding is added for the module whose id is in $r(\mathbf{r}_0)$, i.e., for the module that calls `registerObj(\cdot)`. Parameters w and w' for these procedures are expected respectively in registers \mathbf{r}_7 and \mathbf{r}_8 .

<code>testObj(w, w')</code>	<code>registerObj(w, w')</code>
if $w \notin \text{dom}(\mathcal{G})$ then abort	if $w \in \text{dom}(\mathcal{G})$ then abort
if $\mathcal{G}(w) \equiv (w', _)$ then	$\mathcal{G} + w \mapsto w', r(\mathbf{r}_0)$
return 0	return 0
else return 1	

4.2.2 The Global Call Stack \mathcal{S}

\mathcal{S} tracks all AIL-level function calls. Formally, $\mathcal{S} = \overline{(a)}$.

$\llbracket Sys \rrbracket_{AIL}^{JEM}$ provides the following procedures implemented at entry points: `forwardCall()` and `forwardReturn()`. Both procedures use the following helper functions: `resetFlags(\cdot)` sets flags to 0, `resetRegisters(\cdot)` inputs the registers that need to be reset to 0 and `resetRegistersExcept(\cdot)` inputs the registers that need *not* be reset to 0.

`forwardCall()` reads an address from r_3 and r_4 and forwards the call there. Before forwarding, it stores in \mathcal{S} the address where to return as passed via registers r_0 and r_5 . In order to ensure a correct return, the procedure stores the address of the entry point for `forwardReturn()` (i.e., $3 * \mathcal{N}_w$) in r_5 . Since the caller id is placed in r_0 by the caller-callee authentication mechanism of *PMA*, it cannot be tampered with. This procedure aborts if the module jumping here is the one that jumped here last (if any) or if the address where the call is forwarded is inside $\llbracket Sys \rrbracket_{AIL}^{JEM}$.

```

forwardCall()
-----
If  $\mathcal{S} \neq \emptyset$ 
  Let  $\mathcal{S} = (id, \_); \mathcal{S}'$ 
  if  $r(r_0) == id$  then abort
  if  $r(r_3) == 1$  then abort
  Push  $(r(r_0), r(r_5))$  on  $\mathcal{S}$ 
  Set  $r_5$  to  $3 * \mathcal{N}_w$ 
  resetFlags()
  resetRegisters( $r_0, r_1, r_2$ )
  jmp  $r_3 r_4$ 

```

`forwardReturn()` pops the head of the stack and returns there, setting r_5 to 1 as expected in the case for returns. If the call stack is empty, it aborts to prevent returning when no code was called. If the id of the caller module stored on \mathcal{S} is different from the id of the module jumping to this procedure, it aborts, as it detects a non-well-bracketed execution flow.

```

forwardReturn()
-----
If  $\mathcal{S} = \emptyset$  then abort
Pop  $(id, n)$  from  $\mathcal{S}$  into  $r_2$  and  $r_1$ 
if  $id \neq r(r_0)$  then abort
Set  $r_5$  to 1;
resetFlags()
resetRegistersExcept( $r_1, r_2, r_5, r_6$ )
jmp  $r_1 r_2$  // i.e., jump to address  $(id, n)$ 

```

The calling convention is updated as follows: calls to different modules must go via $\llbracket Sys \rrbracket_{AIL}^{JEM}$, with registers r_3 and r_4 set to the method that needed to be called. Both these procedures do not use registers r_6 onwards, so they do not alter the calling convention regarding parameters and returned values.

4.2.3 Functionality of $\llbracket Sys \rrbracket_{AIL}^{JEM}$

The addition of $\llbracket Sys \rrbracket_{AIL}^{JEM}$ may seem to violate full abstraction as it provides target-level functionality that are not available in JEM [1]. Let us now see why this is not the case.

Procedure `testObj`(\cdot) is analogous to the `instanceof` expression in JEM. This procedure aborts when the object id parameter is not registered in \mathcal{G} i.e., an ill-formed execution of `instanceof`. Aborting in this case lets $\llbracket Sys \rrbracket_{AIL}^{JEM}$ prevent object id guessing (Problem 1).

Procedure `registerObj`(\cdot) is analogous to object creation. This procedure aborts when the object id is already registered; aborting in this case ensures that object ids are globally unique.

Procedure `forwardCall`(\cdot) is analogous to performing a function call. This procedure aborts when the same module performs two calls in a row without there being a return in between or if it performs a call to $\llbracket Sys \rrbracket_{AIL}^{JEM}$ as those behaviours are not available in JEM.

Procedure `forwardReturn`(\cdot) is analogous to returning from a function call. This procedure aborts when a return is not made by the module who was called last. This enforces well-bracketed control flow, preventing Problem 2 as well as external code returning when no method was called.

All the procedures implemented in $\llbracket Sys \rrbracket_{AIL}^{JEM}$ add functionality in AIL that is already available in JEM, so adding $\llbracket Sys \rrbracket_{AIL}^{JEM}$ when linking AIL components does not violate full-abstraction.

4.2.4 $\llbracket Sys \rrbracket_{AIL}^{JEM}$ Definition

Assume the functions provided by $\llbracket Sys \rrbracket_{AIL}^{JEM}$ span memory m , whose size is n addresses. Let the data needed by these functions plus the initialisation of \mathcal{G} and \mathcal{S} span memory m' . $\llbracket Sys \rrbracket_{AIL}^{JEM}$ is always compiled to a module with *id* 1 (since 0 is the id of unprotected code) and it is defined as follows:

$$(m + m'; (1, n, 4); EM_{to}, EM_{ro}, EM_{rc}, EM_{tr}; \emptyset; \emptyset; \emptyset)$$

$\llbracket Sys \rrbracket_{AIL}^{JEM}$ exports a method binding for each procedure it defines; it has no exported object bindings and no required bindings.

- $EM_{to} = \text{testObj} : \text{Obj}(\text{Obj}, \text{Obj})\text{Bool} \mapsto (1, 0)$
- $EM_{ro} = \text{registerObj} : \text{Obj}(\text{Obj}, \text{Obj})\text{Unit} \mapsto (1, \mathcal{N}_w)$
- $EM_{rc} = \text{forwardCall} : \text{Obj}(\cdot)\text{Unit} \mapsto (1, 2 * \mathcal{N}_w)$
- $EM_{tr} = \text{forwardReturn} : \text{Obj}(\cdot)\text{Unit} \mapsto (1, 3 * \mathcal{N}_w)$

$\llbracket \cdot \rrbracket_{AIL}^{JEM}$ knows the offset of all procedures defined in $\llbracket Sys \rrbracket_{AIL}^{JEM}$, so it can perform calls to them. For example a call to `registerObj`(\cdot) is a `jmp rd ri` where $r(\mathbf{r}_d) \mapsto \mathcal{N}_w$ and $r(\mathbf{r}_i) \mapsto 1$.

4.3 The Second Step: `protect(·)`

The `protect(·)` function is a wrapper that takes the memory generated by $\llbracket \cdot \rrbracket$ and adds checks to it, making it secure. This section first presents the required helper functions to compile JEM features: dynamic memory allocation (Section 4.3.1), function calls (Section 4.3.2) and outcalls (Section 4.3.3). The definition of `protect(·)` is provided last (Section 4.3.4).

4.3.1 Dynamic memory allocation

Dynamic memory allocation is the creation of objects at runtime via the `new` expression in JEM. The representation of an object id within a compiled component does not change. To ensure security of object ids when communicated between AIL modules, object ids are no longer just an address in memory, they are symbolic nonces. The latter are called *cross-module object ids* and they are denoted with $\llbracket o \rrbracket_{\text{AIL}}^{\text{JEM}}$. Formally: $\llbracket o \rrbracket_{\text{AIL}}^{\text{JEM}} = \pi$. Symbolic nonces cannot be forged nor guessed, so the format of cross-module object ids addresses Problem 1. Cross-module object ids being a symbolic nonce instead of an address does not disrupt the functionality of compiled JEM code, which functions in the same way as before.

To relate cross-module and internal object ids within a module, masking tables are used [35]. A masking table, indicated with \mathcal{T} , is a bidirectional hash map between internal object id representations and cross-module ones; each module has its own masking table. Formally $\mathcal{T} ::= \overline{w} \mapsto \pi$. Before an (internal) object id w is first passed to external code, it is placed in the table. This process is called *masking*, thus the passed id π is called the *mask*. Denote the retrieval of a mask π with $\mathcal{T}(\pi)$ and the retrieval of an internal object id w with $\mathcal{T}(w)$. Any retrieval causes abortion if the element to be retrieved is not in \mathcal{T} . Adding an internal object id w to a table \mathcal{T} is denoted with $\mathcal{T} + w$; the binding $w \mapsto \pi$, where π is fresh, is added to \mathcal{T} .

Next are the functions used to manage masking tables.

A compiled component must store the encoding of the class it contains for helper functions to rely on. Function `classOf(n)` returns the encoding of the class implemented by the object compiled at address n in the current module. Function `isInternal(w)` takes a class type encoding and returns true if the current module implements that class.

Function `updateMaskingTable(w, n)` inputs a pair of an internal object id and a class type encoding. This function is invoked before releasing an object to external code to add the id of freshly-allocated object to the masking table. This function retrieves the class of the object and then checks if that object id is already in the masking table; if not, it adds the id to the table and registers its cross-module id globally. Retrieving the class object may be necessary, as some arguments may have formal type `Obj`. A crucial part of this function is the call to `registerObj($\mathcal{T}(w), n'$)`, which makes the information that object w , with cross-module id $\mathcal{T}(w)$, implements class n' globally available via $\llbracket Sys \rrbracket_{\text{AIL}}^{\text{JEM}}$.

$$\frac{\text{updateMaskingTable}(w, n)}{\text{if } n \text{ is a class type encoding then} \\ \text{let } n' = \text{classOf}(w) \\ \text{if isInternal}(n') \text{ and if } w \notin \text{dom}(\mathcal{T}) \text{ then} \\ \mathcal{T} + w ; \text{registerObj}(\mathcal{T}(w), n')}$$

Function `loadObjects(\cdot)` loads the internal object ids of the masking indexes that are passed as input into the related register.

$$\frac{\text{loadObjects}(\pi_1, \dots, \pi_n)}{\forall i \in 1..k \quad r(\mathbf{r}_i) \mapsto \mathcal{T}(\pi_i)}$$

Function `maskingTable(EO)` creates the masking table for all exported objects bindings listed in EO .

$$\frac{\text{maskingTable}(o_1 : c_1 \mapsto n_1, \dots, o_k : c_k \mapsto n_k) = \mathcal{T}}{\forall i \in 1..k \quad \text{let } w_i = \text{new } \mathbf{r}_0 \quad \mathcal{T} = n_1 \mapsto w_1, \dots, n_k \mapsto w_k}$$

4.3.2 Function calls

Function calls are calls from outside to within a compiled module. The *PMA* access control policy ensures that these calls can only be calls to entry points. Therefore, an entry point is created for all methods [35]. Function `methodEP(M_t, n)` creates the code to be placed at a method entry point for a method with signature M_t whose implementation is located at address (id, n) in the current module with id id . Following is the pseudo-code of method entry points; notation $n \mapsto \text{code}$ means that n is the address where `code` is located.

$$\frac{\text{methodEP}(t(t_1, \dots, t_k) \rightarrow t', n) =}{\begin{array}{l} \text{If } r(\mathbf{r}_0) \neq 1 \text{ or } r(\mathbf{r}_5) \neq 3 * \mathcal{N}_w \text{ then abort} \\ \text{loadObjects}(r(\mathbf{r}_6), r(\mathbf{r}_7), \dots, r(\mathbf{r}_{6+k})) \\ \text{dynamicTypechecks}(r(\mathbf{r}_6), \langle t \rangle) \\ \text{for } i = 1 \text{ to } k \\ \quad \text{dynamicTypechecks}(r(\mathbf{r}_{6+i}), \langle t_i \rangle) \\ n' \mapsto \text{Set } \mathbf{r}_5 \text{ to } n' + 1 \text{ and jump to module-local address } n \\ \text{updateMaskingTable}(r(\mathbf{r}_6), \langle t' \rangle) \\ \text{maskObjectId}(6) \\ \text{Set } \mathbf{r}_5 \text{ to } 1 \text{ and } \text{jmp} \text{ to the address } (1, 3 * \mathcal{N}_w) \end{array}}$$

Upon jumping to an entry point, a check is made that the jump comes from the $\llbracket Sys \rrbracket_{\text{AIL}}^{\text{JEM}}$ module and that the return address is the `forwardReturn(\cdot)` address (i.e., $3 * \mathcal{N}_w$ in a module whose id is 1). If this is not the case, execution is aborted as some code is trying to bypass $\llbracket Sys \rrbracket_{\text{AIL}}^{\text{JEM}}$. All data needed for this check is known statically. Then, masked object ids are loaded via function `loadObjects(\cdot)`. Only parameters of object type are loaded; if a parameter could not be loaded, the execution is aborted. Once the objects are loaded,

dynamic typechecks are made via function `dynamicTypechecks(·)` (explained below). These checks are located inside the module where class t is compiled, so checking that the current object (\mathbf{r}_6) is of type $\langle t \rangle$ is equivalent to checking that the current object implements the current method. Then the code jumps to the method body located at address n setting \mathbf{r}_5 to the address where that code must return. There, current arguments are placed on the module-local stack alongside other information required by function activation records. When the method body returns (to address $n' + 1$), the masking table is (possibly) updated with the value to be returned (in \mathbf{r}_6) via function `updateMaskingTable(·)` and internal object ids in registers are masked via function `maskObjectId(·)`.

Let us now provide details about the auxiliary functions.

`dynamicTypechecks(·)` ensures that each parameter inhabits its type. `Unit`-typed values are checked to be $\langle \text{unit} \rangle$ and `Bool`-typed ones are checked to be either $\langle \text{true} \rangle$ or $\langle \text{false} \rangle$ [35, 20]. Objects are dynamically typechecked by means of the `testObj(·)` function (as discussed in Section 4.2). If any check fails, the execution is aborted. By checking the existence and types of all parameters, securely compiled code is resilient to Problems 3 and 4.

`dynamicTypechecks(w, n)`

if $n \equiv \langle \text{Unit} \rangle$ then if $w \neq \langle \text{unit} \rangle$ then abort
 if $n \equiv \langle \text{Bool} \rangle$ then if $w \neq \langle \text{true} \rangle$ and $w \neq \langle \text{false} \rangle$ then abort
 if n is a class type and $w \neq \langle \text{null} \rangle$ then
 if `testObj(w, n)` == 1 then abort

Function `maskObjectId(·)` inputs the index of the register to mask and loads a cross-module object id there.

`maskObjectId(n)`

$r(\mathbf{r}_n) \mapsto \mathcal{T}(r(\mathbf{r}_n))$

4.3.3 Outcalls

Outcalls are the dual of function calls, i.e., they are calls from within to outside a module. To allow returning from outcalls, a specific entry point must be created: the *return entry point* [35].

Function `preamble(·)` returns what all code must execute before making an outcall, function `returnEP(·)` returns the code m to be placed at the return entry point.

The *preamble* code is executed before jumping to an external method located at address a , assume this address is communicated via registers \mathbf{r}_0 and \mathbf{r}_1 . Compiled code will execute the `preamble(·)` before jumping outside to ensure the right checks are made. `signatureOf(·)` is used to determine the signature of the compiled method related to a . All method signatures are known statically so their signature encoding can be stored in a table mapping addresses to signature encodings; the table is placed in the data section of the module. This data is communicated to `protect(·)` via the required method bindings

```

preamble() =
  Let  $a$  be the address where the external method is;  $a \equiv (id, w)$ 
  signatureOf( $a$ ) =  $\langle t \rangle(\langle t_1 \rangle, \dots, \langle t_n \rangle) \rightarrow \langle t' \rangle$ 
  storeData( $r(r_6), \langle t' \rangle, r(r_5)$ )
  for  $i = 1$  to  $n$ 
    updateMaskingTable( $r(r_{6+i}), \langle t_i \rangle$ )
    maskObjectId( $6 + i$ )
  resetRegistersExcept( $r_0, \dots, r_{6+n}$ )
  Set  $r(r_3)$  to  $id$  and  $r(r_4)$  to  $w$ 
  Call forwardCall( $\cdot$ )

```

RM returned by $\langle \cdot \rangle$. `preamble()` then calls to `storeData(\cdot)`, which stores the current object r_6 , the expected return type $\langle t' \rangle$, and where to resume the execution after the outcall, an address that is assumed to be passed in r_5 . For any parameter, the masking table is updated with any possible newly created object via function `updateMaskingTable(\cdot)` and their ids are masked with function `maskObjectId(\cdot)`. Then, registers that are not used to convey parameters nor the address where to jump are reset to 0 by function `resetRegistersExcept(\cdot)` in order not to leak information ($\llbracket Sys \rrbracket_{AIL}^{JEM}$ will erase unused registers with index less than 6). This ensures that r_5 contains 0, so $\llbracket Sys \rrbracket_{AIL}^{JEM}$ will return to the return entry point located at address 0 when forwarding the return after this call. Finally the code sets r_3 and r_4 as expected by $\llbracket Sys \rrbracket_{AIL}^{JEM}$, then it jumps to the proxy function for method calls: `forwardCall(\cdot)`.

```

returnEP() =
  0  $\mapsto$  If  $r(r_0) \neq 1$  then abort
          loadData() // access  $w_o, \langle t' \rangle, n'$ 
          loadObjects( $r(r_6)$ )
          dynamicTypechecks( $r(r_6), \langle t' \rangle$ )
          Resume execution from address  $n'$  with current object  $w_o$ 

```

When a return is made, if the module returning is not $\llbracket Sys \rrbracket_{AIL}^{JEM}$ (i.e., a module with id 1), then the code aborts, as some code is trying to bypass $\llbracket Sys \rrbracket_{AIL}^{JEM}$. Then, the current object w_o , the expected return type $\langle t' \rangle$ and the address where to resume execution n' are loaded from the module-local stack via function `loadData(\cdot)` into registers r_7 onwards (since these registers are unused). Finally, the returned value r_6 is checked to be of the expected type. Execution aborts if any check fails, otherwise it resumes within the module with the loaded current object.

4.3.4 The `protect(\cdot)` function

`protect(\cdot)` is formalised in Rule `protect(\cdot) definition`.

To ensure that the compiler is correct, `protect(\cdot)` needs to provide an implementation of `instanceof`, addressing Problem 5 below.

Problem 5 (Using `instanceof`). Consider a module M that contains object o implementing class c . Another module M_1 executes the following code: `instanceof(o : c)`. In order to tell if the test succeeds or not, the code of M_1 must know the class of o . However, with the strong encapsulation provided by PMA, that information resides in the memory of M , which is not accessible by M_1 . ■

To correctly implement cross-module `instanceof`, the class of an object needs to be publicly known, which is a functionality provided by $\llbracket Sys \rrbracket_{AIL}^{JEM}$. When calling `instanceof` on an object whose type is implemented in another module, it suffices to call `testObj(·)` to know if the test succeeds or not. To ensure this happens, the required method binding for `instanceof` is replaced with a method binding for the same symbols to `testObj(·)`. Linking to $\llbracket Sys \rrbracket_{AIL}^{JEM}$ will ensure that those symbols are replaced with the address of `testObj(·)`.

$$\begin{array}{l}
\text{(protect(·) definition)} \\
\llbracket C \rrbracket = (m_c + m_{exit}; m_d; EM; EO; RM + RM_i; RO) \\
EM = m_1 : M_{t1} \mapsto (id, n_1), \dots, m_k : M_{tk} \mapsto (id, n_k) \\
EM' = m_1 : M_{t1} \mapsto a_1, \dots, m_k : M_{tk} \mapsto a_k \\
EO = o_1 : c_1 \mapsto n_1'', \dots, o_j : c_j \mapsto n_j'' \\
EO' = o_1 : c_1 \mapsto \mathcal{T}(n_1''), \dots, o_j : c_j \mapsto \mathcal{T}(n_j'') \\
RM_i = instanceof : \text{Obj}(\text{Obj}, \text{Obj})\text{Bool} \mapsto (\iota_{inst}, \sigma_{inst}) \\
RM'_i = testObject : \text{Obj}(\text{Obj}, \text{Obj})\text{Bool} \mapsto (\iota_{inst}, \sigma_{inst}) \\
s = (id, n', k + 1) \quad \text{maskingTable}(EO) = \mathcal{T} \\
m_m = \text{implementationOf}(\mathcal{T}) \quad m_r = \text{returnEP}() \\
m_{ce} = \text{extraCode}() \quad m_{de} = \text{extraData}() \\
\forall i \in 1..k \quad a_i = (id, i \cdot \mathcal{N}_w) \quad m_i = \text{methodEP}(M_{ti}, n_i) \\
m'_{exit} = \text{append}(m_{exit}, \text{preamble}(\cdot)) \\
m_{code} = m_1 + \dots + m_k + m_r + m_c + m'_{exit} + m_{ce} \\
n' = |m_{code}| \quad m = m_{code} + m_d + m_{de} + m_m \\
\hline
\text{protect}(\llbracket C \rrbracket) = m; s; EM'; EO'; RM + RM'_i; RO
\end{array}$$

`protect(·)` calculates the memory layout s based on the functionality listed in EM . The memory created by `protect(·)` contains the following functionality. Each method exported to external code is given an entry point with code created with the `methodEP(·)` function. Function `returnEP(·)` yields memory m_r for the return entry point. The code section of m is completed with the `extraCode(·)`, containing the implementation of all the helper functions described above (e.g., `dynamicTypechecks(·)`, `resetFlagsAndRegs(·)` etc.) and with m_c , the code generated by $\llbracket \cdot \rrbracket$. Additionally, the code section appends the `preamble(·)` procedure at the exit-point code provided by $\llbracket \cdot \rrbracket$ to ensure that it is always executed before exiting the module. After the code section, m contains `extraData(·)`, i.e., all the data needed by the helper functions (e.g., the encodings of method signatures and of types) and by m_d , the data generated by $\llbracket \cdot \rrbracket$. Finally the data section contains the masking table m_m , which is obtained with function `maskingTable(·)`; as memories are infinite, the data section of a module has no boundaries.

What `protect(·)` returns exports the same methods and objects as $\llbracket \cdot \rrbracket$. The former are bound to entry points and the latter have their ids masked.

4.4 The Secure Linker: `link(·)`

This section presents function `link(·)`, which inputs and returns `AIL` modules; it is formalised as follows:

$$\begin{aligned} \text{link}(P, P') &= M_1 \uplus \dots \uplus M_n \uplus M'_1 \uplus \dots \uplus M'_m \\ \text{where } P &\equiv M_1 \uplus \dots \uplus M_n \\ \text{and } P' &\equiv M'_1 \uplus \dots \uplus M'_m \end{aligned}$$

Modules M_i are obtained through calls to $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$, they all are a pair of a compiled `JEM` class and an instance of $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$. Informally, the \uplus operator does the following:

- it performs `AIL`-level joining of modules ($+$, presented in Section 2.2.2) ensuring that only one occurrence of $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$ is present in the resulting component;
- it initialises the resulting $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$ table \mathcal{G} with all exported object bindings for all M_i , so that static objects are registered in \mathcal{G} .

5 Discussion

This section presents how to extend $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ to a source language supporting object-orientation (Section 5.1) and how to support multi-register data (Section 5.2).

5.1 Supporting Object-Orientation

`JEM` can be made object-oriented by adding support for interfaces, inheritance and dynamic dispatch. We suggest doing this by adding interfaces to `JEM`, making class types private to a component and only using interface types in the types of cross-component method calls and returns [25, 35]. Since different `JEM` components can implement the same interface, a number of concerns arise. Nevertheless, we believe the concerns can be adequately addressed and this section also describes a possible change to our compiler which we believe is a way to solve the concerns.

Problem 6 (Module id at the target level). *Consider two modules M_1 and M_2 containing the compilation of two classes that implements the same interface i . Code in another module M could input objects of that interface. However, M cannot be sure of where the object id is located unless it indicates its module id. In fact, all that M knows when receiving an argument is the type i , but both M_1 and M_2 implement it.* ■

To address Problem 6, cross-module object ids need to state the module id where they reside; formally $\llbracket o \rrbracket_{\text{AIL}}^{\text{JEM}} = \pi, id$. Unless AIL is extended to merge this information in nonces, object ids span multiple words (Section 5.2 describes how to securely compile them).

Often, object-oriented languages implements dynamic dispatch as vtables that are located at the address where an object is compiled. With PMA, M_1 has no access to the vtable of o if o is allocated in M_2 . The single-module version of AIL [35] used entry points as a vtable but this does not hold when multiple modules are considered.

Problem 7 (Dynamic dispatch). *Consider two modules M_1 and M_2 containing the compilation of two classes implementing the same interface Bank . Module M_1 also implements interface Account while module M_2 also implements interface Currency . Since methods need to be sorted alphabetically based on their namespaces (which include interface names), methods for Bank in M_1 will have different entry points than the same methods in M_2 .*

A module M interacting with M_1 and M_2 however should not need to know the full specification of which component contains which interfaces, as this would defeat the purpose of object orientation. All that M knows is that somewhere outside its memory, interface Bank is implemented. When receiving an object from M_1 or from M_2 that implements the Bank interface, M needs to be able to calculate where to jump in order to call methods on it. ■

To address Problem 7, a single method entry point is created at address \mathcal{N}_w instead of an entry point per method implemented in a component. That entry point serves as a dynamic dispatch entry point. All cross-module method calls update the calling convention to use \mathbf{r}_6 as a container for the encoding of the method to be called. Formally, indicate a method encoding as follows $\langle m \rangle = n$.

When a module receives an object id of the form π, id , it can call method m on it by jumping to address $(id, 0)$ and setting \mathbf{r}_6 to $\langle m \rangle$. Parameters and the current object are then passed via registers \mathbf{r}_7 onwards.

The dynamic dispatch entry point must perform a new check: \mathbf{r}_6 must be a valid method encoding. All methods implemented in interfaces are known statically, so a module can save this information in its memory to encode this check. If this check succeeds, the execution continues by dispatching to the checks for method entry points of Section 4.3.

5.2 Supporting Multi-Register Data

In some cases, securely-compiled code needs to communicate data that spans multiple registers. For example, the source language could be extended to support computation on complex numbers, or the format of object ids could vary (as discussed in Section 5.1). Communicating data that spans multiple registers can be done by extending AIL with cryptographic functions (either in the form of instructions, as presented here, or as module-internal procedures).

First, all modules must have a public and a private key, all modules know each other's public keys but the private one is confidential to the module [31].

Second, cryptographic functions for signing and verifying signatures are needed.

<code>sign r_s r_e r_k r_d</code>	Sign all registers from r_s to r_e with the (private) key found in register r_k and place the result in register r_d .
<code>verify r_s r_e r_k r_d</code>	Set the ZF according to whether the signature found in register r_d has been created for all registers from r_s to r_e with the dual key of that found in register r_k .

Consider the case of multi-register object ids of Section 5.1; cross-module object ids need to be changed into a triplet, so they span three registers. The triplet consists of: a masking index (as in Section 4.3), a type encoding and a signature of the two; the signature prevents Problem 8 below; Formally: $\llbracket o \rrbracket_{\text{AIL}}^{\text{JEM}} = \pi, n, w$.

Problem 8 (Object-id shuffling). *Consider two objects o_1 and o_2 residing in two modules M_1 and M_2 implementing two different classes. Without signatures, those objects would have the following cross-module object ids: $\llbracket o_1 \rrbracket_{\text{AIL}}^{\text{JEM}} = \pi_1, n_1$ and $\llbracket o_2 \rrbracket_{\text{AIL}}^{\text{JEM}} = \pi_2, n_2$. An attacker can forge new object ids as follows: π_1, n_2 and π_2, n_1 . Another module M_3 receiving the forged objects from the attacker has no way to tell whether they are forged by inspecting them.* ■

With the signature in place, the object ids of o_1 and o_2 change as follows: $\llbracket o_1 \rrbracket_{\text{AIL}}^{\text{JEM}} = \pi_1, n_1, w_1$ and $\llbracket o_2 \rrbracket_{\text{AIL}}^{\text{JEM}} = \pi_2, n_2, w_2$. An attacker can still forge object ids by creating π_1, n_2, w_2 , but M_3 can verify the signature part of the triplet, therefore finding out when forgery has taken place.

With this approach, some auxiliary functions need to be changed. Function `loadObjects(\cdot)` needs to verify that the object ids of all externally-located modules have been signed with the proper key and that the key corresponds to the module that implements the class n mentioned in the object id. As a module implements a single class, the type encoding tells a module which public key to use for the verification. Function `updateMaskingTable(\cdot)` needs to ensure that communicated cross-module object id are signed.

Concerning $\llbracket Sys \rrbracket_{\text{AIL}}^{\text{JEM}}$, `registerObj(\cdot)` is not needed and the pseudo code inserted by `protect(\cdot)` does not call it. Procedure `testObj(\cdot)` can be implemented locally in a module, since due to the new cross-module object ids the type of an object is part of its id.

6 Full-Abstraction and Modularity of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$

As stated in Section 1, a compiler is fully-abstract if it preserves and reflects contextual equivalence of source and target components. This section briefly discusses how contextual equivalence is reflected (Section 6.1) and preserved (Section 6.2) for $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$. Then it concludes by presenting the proof sketch of full-abstraction and modular full-abstraction of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ (Section 6.3).

6.1 $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ Reflects Contextual Equivalence

Proving that $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ reflects contextual equivalence (Theorem 1) is analogous to proving that it is correct and adequate.

Theorem 1 ($\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ preserves behaviour). $\forall \mathcal{C}_1, \mathcal{C}_2. \llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \simeq_{\text{ctx}} \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}} \Rightarrow \mathcal{C}_1 \simeq_{\text{ctx}} \mathcal{C}_2$.

Intuitively, given Assumption 4, this holds because neither $\text{protect}(\cdot)$ nor the addition of $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$ change the semantics of compiled programs.

6.2 $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ Preserves Contextual Equivalence

Proving that $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ preserves contextual equivalence (Theorem 2) is analogous to proving that $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is secure. Theorem 2 intuitively states that JEM abstractions are preserved in the AIL output $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ produces.

Theorem 2 ($\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is secure). $\forall \mathcal{C}_1, \mathcal{C}_2. \mathcal{C}_1 \simeq_{\text{ctx}} \mathcal{C}_2 \Rightarrow \llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \simeq_{\text{ctx}} \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}$.

For Theorem 2 we proceed as follows. First, we devise a notion of trace equivalence $\underline{\simeq}$ for securely-compiled components; this is a slight adaptation of a similar trace semantics for the single-module version of AIL [36]. Intuitively the trace semantics describes the behaviour of a set of modules with sets of traces, i.e., concatenation of actions such as call and return. Most importantly, trace semantics lets us disregard contexts when reasoning about modules behaviour. We assume that $\underline{\simeq}$ is equivalent to \simeq_{ctx} , so the two notions can be exchanged.

Assumption 5 (Trace semantics coincides with contextual equivalence for securely compiled JEM components). $\forall \mathcal{C}_1, \mathcal{C}_2. \llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \simeq_{\text{ctx}} \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}} \iff \llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \underline{\simeq} \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}$.

Then we re-state Theorem 2 in contrapositive form:

$$\forall \mathcal{C}_1, \mathcal{C}_2. \llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \not\underline{\simeq} \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}} \Rightarrow \mathcal{C}_1 \not\simeq_{\text{ctx}} \mathcal{C}_2$$

To achieve $\mathcal{C}_1 \not\simeq_{\text{ctx}} \mathcal{C}_2$ we need to show (by negating Definition 1) that there exist a context that, wlog, terminates with \mathcal{C}_1 and diverges with \mathcal{C}_2 . This context is said to *differentiate* between \mathcal{C}_1 and \mathcal{C}_2 . For this, we devise an algorithm $\langle\langle \cdot \rangle\rangle$ that can always generate such a differentiating context given two JEM components whose compiled counterparts are trace-inequivalent [17, 35, 36]. Since $\langle\langle \cdot \rangle\rangle$ is sketched to be correct (Theorem 3), we can use it to witness that two JEM components are contextually-inequivalent if their compiled counterparts are trace-inequivalent.

Theorem 3 (Algorithm correctness). $\forall \mathcal{C}_1, \mathcal{C}_2, \bar{\alpha}_1 \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}), \bar{\alpha}_2 \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}), \alpha_1 \neq \alpha_2, \langle\langle \mathcal{C}_1, \mathcal{C}_2, \bar{\alpha}_1, \bar{\alpha}_2 \rangle\rangle = \mathbb{C} \text{ such that } \mathbb{C}[\mathcal{C}_1] \uparrow \iff \mathbb{C}[\mathcal{C}_2] \uparrow$.

6.3 Full-Abstraction and Modularity

By the theorems of Sections 6.1 and 6.2, $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is fully-abstract (Theorem 4).

Theorem 4 ($\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is fully-abstract). $\forall \mathcal{C}_1, \mathcal{C}_2. \mathcal{C}_1 \simeq_{\text{ctx}} \mathcal{C}_2 \iff \llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \simeq_{\text{ctx}} \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}$.

The novel result we are after for $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is *modular full-abstraction*, i.e. components can be compiled separately and linked together afterwards without compromising security. Indicate the linking of two AIL components P_1 and P_2 as $\text{link}(P_1, P_2)$. This definition is derived from Ahmed’s definition of horizontal compiler compositionality [5]; it states that we can create source-level components by joining an arbitrary number of them and by compiling them individually and then linking the result. Additionally, we prove this result for arbitrary target-level components P and P' that are equivalent to securely-compiled ones. P and P' can be seen as hand-optimised versions of \mathcal{C}_2 and \mathcal{C}_4 that respect the behaviour imposed by $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$. Formally, we have that Corollary 1 is a corollary of Theorem 4.

Corollary 1 (Modular full-abstraction). $\forall \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4. \forall P. \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}} \simeq_{\text{ctx}} P, \forall P'. \llbracket \mathcal{C}_4 \rrbracket_{\text{AIL}}^{\text{JEM}} \simeq_{\text{ctx}} P', \mathcal{C}_1; \mathcal{C}_2 \simeq_{\text{ctx}} \mathcal{C}_3; \mathcal{C}_4 \iff \text{link}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}, P) \simeq_{\text{ctx}} \text{link}(\llbracket \mathcal{C}_3 \rrbracket_{\text{AIL}}^{\text{JEM}}, P')$.

7 Related Work

Secure compilation through full-abstraction was pioneered by Abadi [1] and successfully applied to many different settings [35, 20, 13, 7, 6, 4, 24, 18]. Parrow proved which conditions must hold in source and target languages to provide a fully-abstract compiler between the two [32]. Gorla and Nestmann concluded that full-abstraction is meaningful when it entails properties like security [21].

A large body of research provided secure compilers for a variety of languages with different language features. Three works are closely related to the present one. The first one is the fully-abstract compilation scheme of single-module code to single-module PMA of Patrignani *et al.* [35], where linking is not explicitly considered. The second one is the secure compiler targeting an extension of the PUMP machine (a tag-based assembly-level architecture that enforces micro-policies with each instruction) [9] by Juglaret and Hritcu [26]. This work considers a very similar source language, so it incurs in very similar problems to what discussed here. Intuitively, Juglaret and Hritcu use tags to create PMA-like modules (with their local stacks) where jumps can only be done at specific addresses. By relying on a very different architecture, their solution differs significantly from ours. For example, to address Problem 2, they rely on linear tags for return addresses. Their tags also capture type information, so that dynamic typechecks (as needed for Problem 3) are made based on tags. As stated in Section 1, the main difference between the two works is in the architecture they target: while SGX-like PMA is readily available, it may be a while before PUMP-like machine hit the market. The third closest secure compilation result is the (probabilistic) fully-abstract compiler to ASLR-enhanced

target languages, which also does not explicitly consider linking [4, 24]. ASLR prevents some linking problems (e.g., object guessing) but not all of them (e.g., call stack shortcutting). We expect that ASLR-based secure compilation can be made resilient to all linking related attacks by adopting analogous counter-measures to those described in this paper.

Most secure compilation works adopt the fully-abstract compilation notion of Abadi [1]. These works achieve security by relying on type systems for the target language [7, 6, 20], cryptographic primitives [14, 15, 2] and the already mentioned ASLR [4, 24] and PMA [35]. Additionally, certain works provide secure compilers by means of type-preserving compilers [11, 10, 41], though they require the target language to be well-typed. Of all these works, only Abadi *et al.* [2] consider multiple modules, but in a distributed setting rather than on a single machine, so that presents different vulnerabilities than the ones considered in this paper.

8 Conclusion and Future Work

This paper presented a secure compilation scheme from JEM, an object-based imperative language to AIL, an untyped assembly language enhanced with PMA. Because AIL explicitly deals with linking, the secure compiler developed in this paper faces a number of threats that no previous work considers. This paper formalised the compiler $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ and explained how it withstands these new threats. Finally, it presented how $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$ is fully-abstract and modular, so that the additional threats arising from linking cannot be exploited by a malicious attacker.

The authors foresee a number of future research trajectories for this work: integrating a garbage collector with securely compiled programs, supporting secure compilation for concurrent programs and developing secure compilation schemes for emerging security architectures such as capability machines [43, 42].

Acknowledgements. The authors would like to thank Amal Ahmed, Cătălin Hrițcu, Yannis Juglaret, Raoul Strackx and the anonymous reviewers for useful comments on an earlier version of this paper. This work has been supported in part by the Intel Labs University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CENTRE).

Appendix Introduction

The appendix contains formalisation of the JEM (Appendix A) and ALL (Appendix B) languages, the formalisation of the algorithm $\llbracket \cdot \rrbracket$ (Appendix C) and proofs and proof sketches of the paper (Appendix D).

A JEM

The formalisation of JEM borrows extensively from that of Java Jr. [25] and from the single-component version of it [35, 34].

A.1 Static Semantics

A rule of the form $\mathcal{P} \vdash \bar{E}$ is intended to be $\mathcal{P} \vdash E_1, \dots, \mathcal{P} \vdash E_n$ where $\bar{E} = E_1, \dots, E_n$. Denote a stack of variable bindings with $\bar{S} ::= \overline{x : t}$

$$\begin{array}{c}
 \text{(Type-programs)} \\
 \frac{\mathcal{P} = C_1, \dots, C_n, \quad \forall i \in 1..n \quad \mathcal{P} \vdash C_i}{C_i = \text{import } \bar{I}; \bar{X}; \text{ class } c\{K \ \bar{F}_t \ \bar{M}\}; \bar{O} \quad \mathcal{P} \vdash \bar{I} \quad \mathcal{P} \vdash \bar{X}}{\vdash \mathcal{P} : \text{prg}} \\
 \\
 \text{(Interface-checking)} \\
 \frac{\mathcal{P} = C_1, \dots, C_n, \quad \exists i \in 1..n. \\
 C_i = \text{import } \bar{I}; \bar{X}; \text{ class } c\{K \ \bar{F}_t \ \bar{M}M'\}; \bar{O} \\
 \bar{M}_t = M_{t_1}, \dots, M_{t_m} \quad \bar{M} = M_1, \dots, M_m \\
 \forall j \in 1..m. \quad M_j \vdash M_{t_j} \quad c \text{ is unique in } \mathcal{P}}{\mathcal{P} \vdash \text{class-decl } c\{\bar{M}_t\}} \\
 \\
 \text{(Method-checking)} \\
 \frac{M = \text{public } m(\bar{x}) : M_t \{ \text{return } E; \} \quad M_t = m : t(\bar{t}) \rightarrow t'}{M \vdash M_t} \\
 \\
 \text{(Extern-checking)} \\
 \frac{\mathcal{P} \vdash o : c}{\mathcal{P} \vdash \text{obj-decl } o : c} \\
 \\
 \text{(Type-class)} \\
 \frac{\mathcal{P} \vdash \bar{I} \quad \mathcal{P} \vdash \bar{X} \quad \mathcal{P}, c, \bar{F}_t \vdash K \\
 \mathcal{P}; c \vdash \bar{M} \quad \mathcal{P}, c, \bar{F}_t, \bar{O} \vdash \bar{O} \quad c \text{ is unique in } \mathcal{P}}{\mathcal{P} \vdash \text{import } \bar{I}; \bar{X}; \text{ class } c\{K \ \bar{F}_t \ \bar{M}\}; \bar{O}} \\
 \\
 \text{(Type-object)} \\
 \frac{o \text{ is unique in } \mathcal{P} \quad \bar{F}_t = f_1 : t_1, \dots, f_n : t_n, \\
 \bar{F} = \text{private } f_1 = v_1, \dots, \text{private } f_n = v_n \\
 \forall i \in 1..n. \quad \mathcal{P} \vdash v_i : t_i}{\mathcal{P}, c, \bar{F}_t, \bar{O} \vdash \text{object } o : c\{\bar{F}\}} \\
 \\
 \text{(Type-unit)} \\
 \frac{}{\mathcal{P} \vdash \text{unit} : \text{Unit}} \\
 \\
 \text{(Type-bool)} \\
 \frac{v \equiv \text{true} \vee v \equiv \text{false}}{\mathcal{P} \vdash v : \text{Bool}} \\
 \\
 \text{(Type-int)} \\
 \frac{v \in \mathbb{N}}{\mathcal{P} \vdash v : \text{Int}}
 \end{array}$$

$$\begin{array}{c}
\text{(Type-obj)} \\
\frac{\mathcal{P} \vdash t \quad \mathcal{P} = C_1, \dots, C_n, \quad \exists i \in 1..n. \quad C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \quad O = O_1, \dots, O_m, \quad \exists j \in 1..m. \quad O_j = \text{object } o : c\{\overline{F}\}}{\mathcal{P} \vdash v : t} \\
\text{(Type-constructor)} \\
\frac{\overline{F}_t = f_1 : t_1, \dots, f_n : t_n \quad \overline{f} = f_1, \dots, f_n \quad \overline{t} = t_1, \dots, t_n}{\mathcal{P}, c, \overline{F}_t \vdash c(\overline{f} : \overline{t}) \ \{\text{this}.\overline{f}' = \overline{f}\}} \\
\text{(Type-method)} \\
\frac{\overline{x} = x_1, \dots, x_n \quad M_t = m : t(\overline{t}) \rightarrow t' \quad \overline{t} = t_1, \dots, t_n \quad \mathcal{P}; x_1 : t_1, \dots, x_n : t_n; c \vdash E : t'; \overline{S}' \quad m \text{ is unique in } \mathcal{P} \quad \forall i \in 1..n \quad x_i \text{ is unique in } \overline{x}}{\mathcal{P}; c \vdash \text{public } m(\overline{x}) : M_t \ \{\text{return } E; \}} \\
\text{(Type-expr-var)} \\
\frac{x : t \in \overline{S}}{\mathcal{P}; \overline{S}; c \vdash x : t; \emptyset} \\
\text{(Type-expr-field)} \\
\frac{\mathcal{P}; \overline{S}; c \vdash E : c; \emptyset \quad \mathcal{P} = C_1, \dots, C_n, \quad \exists i \in 1..n. \quad C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \quad f : t \in \overline{F}_t}{\mathcal{P}; \overline{S}; c \vdash E.f : t; \emptyset} \\
\text{(Type-expr-field-update)} \\
\frac{\mathcal{P}; \overline{S}; c \vdash E : c; \emptyset \quad \mathcal{P}; \overline{S}; c \vdash E' : t; \emptyset \quad \mathcal{P} = C_1, \dots, C_n, \quad \exists i \in 1..n. \quad C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \quad f : t \in \overline{F}_t}{\mathcal{P}; \overline{S}; c \vdash E.f = E' : \text{Unit}; \emptyset} \\
\text{(Type-expr-method)} \\
\frac{\mathcal{P}; \overline{S}; c \vdash E : t; \emptyset \quad \mathcal{P}; \overline{S}; c \vdash \overline{E} : \overline{t}; \emptyset \quad \mathcal{P} = C_1, \dots, C_n, \quad \exists i \in 1..n. \quad C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c'\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \quad \overline{M} = M_1, \dots, M_m \quad \exists i \in 1..m. \quad M_m = \text{public } m(\overline{x}) : M_t \ \{\text{return } E''; \} \quad M_t = m : t(\overline{t}) \rightarrow t'}{\mathcal{P}; \overline{S}; c \vdash E.m(\overline{E}) : t'; \emptyset} \\
\text{(Type-expr-op)} \\
\frac{\mathcal{P}; \overline{S}; c \vdash E : t; \emptyset \quad \mathcal{P}; \overline{S}; c \vdash E' : t'; \emptyset \quad \text{op} : t \times t' \rightarrow t''}{\mathcal{P}; \overline{S}; c \vdash E \text{ op } E' : t''; \emptyset} \\
\text{(Type-expr-exit)} \\
\frac{\mathcal{P}; \overline{S}; c \vdash E : t; \emptyset}{\mathcal{P}; \overline{S}; c \vdash \text{exit } E : t; \emptyset} \\
\text{(Type-expr-sequence)} \\
\frac{\mathcal{P}; \overline{S}; c \vdash E : t; \overline{S}' \quad \mathcal{P}; \overline{S}, \overline{S}'; c \vdash E; E' : t'; \overline{S}''}{\mathcal{P}; \overline{S}; c \vdash E; E' : t'; \overline{S}', \overline{S}''} \\
\text{(Type-expr-newvar)} \\
\frac{x \notin \text{dom}(\overline{S}) \quad \mathcal{P}; \overline{S}; c, x : t; c \vdash E : t; \emptyset}{\mathcal{P}; \overline{S}; c \vdash \text{var } x : t = E : \text{Unit}; x : t}
\end{array}$$

$$\begin{array}{c}
\text{(Type-expr-if)} \\
\frac{\mathcal{P}; \overline{S}; c \vdash E : \text{Bool}; \emptyset \quad \mathcal{P}; \overline{S}; c \vdash E' : t; \emptyset \quad \mathcal{P}; \overline{S}; c \vdash E'' : t; \emptyset}{\mathcal{P}; \overline{S}; c \vdash \text{if } (E) \{E'\} \text{ else } \{E''\} : t; \emptyset} \\
\text{(Type-expr-new)} \\
\frac{\mathcal{P}; \overline{S}; c \vdash E : \overline{t}; \emptyset \quad \mathcal{P} = C_1, \dots, C_n, \quad \exists i \in 1..n. \\ C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \quad K = c(\overline{f} : \overline{t}) \{\text{this}.\overline{f}' = \overline{f}\}}{\mathcal{P}; \overline{S}; c \vdash \text{new } c(\overline{X}) : c; \emptyset} \\
\text{(Type-expr-val)} \quad \text{(Type-expr-this)} \\
\frac{\exists t. \quad \mathcal{P}; \overline{S}; c \vdash v : t}{\mathcal{P}; \overline{S}; c \vdash v} \quad \frac{}{\mathcal{P}; \overline{S}; c \vdash \text{this} : c}
\end{array}$$

Contexts for the evaluation of expressions are defined as follows: $\mathbb{E} ::= [\cdot] \mid \mathbb{E}.m(\overline{X}) \mid o.m(\overline{v}, \mathbb{E}, \overline{E}) \mid \mathbb{E}.f \mid \mathbb{E}.f = E \mid v.f = \mathbb{E} \mid \text{new } c(\overline{v}, \mathbb{E}, \overline{X}) \mid \text{if}(\mathbb{E})\{E\}\text{else}\{E\} \mid \mathbb{E}; E \mid \text{var } x : t = \mathbb{E} \mid \text{return } \mathbb{E} \mid \mathbb{E} \text{ op } E \mid v \text{ op } \mathbb{E} \mid \text{exit } \mathbb{E} \mid \text{instanceof}(\mathbb{E}, c)$.

Configurations for the dynamic semantics: $k ::= (\mathcal{P}; \overline{B} \vdash \mathbb{E}[E])$.

\overline{B} is a stack of stacks $B ::= \overline{x} \mapsto \overline{v}$, lookup and update is always done to the top stack of the stack.

Transitions of the static semantics $\rightarrow \subseteq k \times k$.

Initial state for a program $\mathcal{P} = \mathcal{P}; \emptyset \vdash [\text{main}.\text{main}()]$.

Define plugging for contextual equivalence as follows: if $\mathbb{C} = \mathcal{C}$, then $\mathbb{C}[\mathcal{C}'] = \mathcal{C}, \mathcal{C}'$ if $\mathbb{C} \cap \mathcal{C}'$. The initial state of $\mathbb{C}[\mathcal{C}']$ is $\mathcal{C}, \mathcal{C}'; \emptyset \vdash [\text{main}.\text{main}()]$. Assume a context always define the *main* method and the *main* class.

$$\begin{array}{c}
\text{(JEM-eval-field-lookup)} \\
\frac{\mathcal{P} = C_1, \dots, C_n, \quad \exists i \in 1..n. \quad C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \\ \overline{O} = O_1, \dots, O_m, \quad \exists j \in 1..m. \quad O_j = \text{object } o : c\{\overline{F}\} \quad \text{private } f = v \in \overline{F}}{\mathcal{P}; \overline{B} \vdash \mathbb{E}[o.f] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[v]} \\
\text{(JEM-eval-field-update)} \\
\frac{\mathcal{P} = C_1, \dots, C_n, \quad \exists i \in 1..n. \\ C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \\ \overline{O} = O_1, \dots, O_m, \quad \exists j \in 1..m. \\ O_j = \text{object } o : c\{\overline{F}\} \quad F = F_1, \dots, F_l \\ \exists h \in 1..l. \quad F_h = \text{private } f = v' \quad F'_h = \text{private } f = v \\ \overline{F}' = F_1, \dots, F_{h-1}, F'_h, F_{h+1}, \dots, F_l \quad O'_j = \text{object } o : c\{\overline{F}'\} \\ \overline{O}' = O_1, \dots, O_{j-1}, O'_j, O_{j+1}, \dots, O_m \\ C'_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O}' \\ \mathcal{P}' = C_1, \dots, C_{i-1}, C'_i, C_{i+1}, \dots, C_n}{\mathcal{P}; \overline{B} \vdash \mathbb{E}[o.f = v] \rightarrow \mathcal{P}'; \overline{B} \vdash \mathbb{E}[\text{unit}]} \\
\text{(JEM-eval-method-call)} \\
\frac{\mathcal{P} = C_1, \dots, C_n \quad \exists i \in 1..n. \quad C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \\ \overline{O} = O_1, \dots, O_m \quad \exists j \in 1..m. \quad O_j = \text{object } o : c\{\overline{F}\} \\ \overline{M} = M_1, \dots, M_k \quad \exists h \in 1..k. \quad M_k = \text{public } m(\overline{x}) : M_t \{\text{return } E; \}}{\mathcal{P}; \overline{B} \vdash \mathbb{E}[o.m(\overline{v})] \rightarrow \mathcal{P}; \emptyset; \overline{B} \vdash \mathbb{E}[E[o/\text{this}][\overline{v}/\overline{x}]]}
\end{array}$$

$$\begin{array}{c}
\text{(JEM-eval-method-return)} \\
\hline
\mathcal{P}; \overline{B}; \overline{B} \vdash \mathbb{E}[\text{return } v] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[v] \\
\text{(JEM-eval-new)} \\
\begin{array}{l}
\mathcal{P} = C_1, \dots, C_n \quad \exists i \in 1..n. \quad C_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \\
\overline{O} = O_1, \dots, O_m \quad \overline{F}_t = \overline{f} : t \quad O = \text{object } o : c\{\overline{f} = \overline{v}\} \quad o \text{ is fresh in } \mathcal{P} \\
\overline{O}' = O_1, \dots, O_m, O \quad C'_i = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O}' \\
\mathcal{P}' = C_1, \dots, C_{i-1}, C'_i, C_{i+1}, \dots, C_n
\end{array} \\
\hline
\mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{new } c(\overline{v})] \rightarrow \mathcal{P}'; \overline{B} \vdash \mathbb{E}[o] \\
\text{(JEM-eval-if-true)} \\
\hline
\mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{if } (\text{true}) \{E_t\} \text{ else } \{E_f\}] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[E_t] \\
\text{(JEM-eval-if-false)} \\
\hline
\mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{if } (\text{false}) \{E_t\} \text{ else } \{E_f\}] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[E_t] \\
\text{(JEM-eval-instanceof-true)} \\
\begin{array}{l}
\exists C \in \mathcal{P}. \quad C = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \\
\exists O \in \overline{O}. \quad O = \text{object } v : c\{\overline{F}\}
\end{array} \\
\hline
\mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{instanceof}(v : c)] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{true}] \\
\text{(JEM-eval-instanceof-false)} \\
\begin{array}{l}
\nexists C \in \mathcal{P}. \quad C = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \\
\exists O \in \overline{O}. \quad O = \text{object } v : c\{\overline{F}\}
\end{array} \\
\hline
\mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{instanceof}(v : c)] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{false}] \\
\text{(JEM-eval-local-var)} \\
\hline
\mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{var } x : t = v] \rightarrow \mathcal{P}; \overline{B} + (x \mapsto v) \vdash \mathbb{E}[v] \\
\begin{array}{cc}
\text{(JEM-eval-lookup-var)} & \text{(JEM-eval-op)} \\
\overline{B}(x) = v & v \text{ op } v' = v''
\end{array} \\
\hline
\mathcal{P}; \overline{B} \vdash \mathbb{E}[x] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[v] & \mathcal{P}; \overline{B} \vdash \mathbb{E}[v \text{ op } v'] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[v''] \\
\text{(JEM-eval-exit)} & \text{(JEM-eval-concatenation)} \\
\hline
\mathcal{P}; \overline{B} \vdash \mathbb{E}[\text{exit } v] \rightarrow \mathcal{P}; \overline{B} \vdash v & \mathcal{P}; \overline{B} \vdash \mathbb{E}[v; E] \rightarrow \mathcal{P}; \overline{B} \vdash \mathbb{E}[E]
\end{array}$$

Define compatibility as follows:

$$\begin{array}{c}
\text{(Component satisfaction)} \\
\frac{\mathcal{C} \Vdash \mathcal{C}' \quad \mathcal{C}' \Vdash \mathcal{C}}{\mathcal{C} \frown \mathcal{C}'}
\end{array}$$

Define a component to satisfy another as $\mathcal{C}' \Vdash \mathcal{C}$. Formally:

$$\begin{array}{cc}
\text{(Component satisfaction base)} & \text{(Component satisfaction inductive)} \\
\frac{}{\mathcal{C}' \Vdash \emptyset} & \frac{\mathcal{C}' \Vdash C_1 \quad \mathcal{C}' \Vdash \mathcal{C}}{\mathcal{C}' \Vdash C_1; \mathcal{C}}
\end{array}$$

$$\begin{array}{c}
\text{(Component satisfaction single)} \\
\hline
\begin{array}{l}
\mathcal{C} = C_1, \dots, C_n \quad C = \text{import } \overline{I}; \overline{X}; \text{ class } c\{K \ \overline{F}_t \ \overline{M}\}; \overline{O} \\
\overline{I} = I_1, \dots, I_m \quad \overline{X} = X_1, \dots, X_k \\
\forall i \in 1..m, \exists j \in 1..n. \quad I_i = \text{class-decl } i\{\overline{M}_t\} \\
C_j = \text{import } \overline{I}'; \overline{X}'; \text{ class } i\{K' \ \overline{F}_t' \ \overline{M}'\}; \overline{O}' \quad \overline{M}' \vdash \overline{M}_t \\
\forall g \in 1..k, \exists h \in 1..n. \quad X_g = \text{obj-decl } o : c; \\
C_h = \text{import } \overline{I}''; \overline{X}''; \text{ class } c\{K'' \ \overline{F}_t'' \ \overline{M}''\}; \overline{O}'' \quad \overline{O}'' = O_1'', \dots, O_l'' \\
\exists h \in 1..l. \quad O_h'' = \text{object } o : c\{\overline{F}\}
\end{array} \\
\hline
\mathcal{C} \Vdash C
\end{array}$$

A component satisfies a class if all import declarations in the class are implemented in the component. So, for all interfaces imported in the class, a class exists in the component with the same name and at least the same required methods. For all externs imported, an object exists with the same name and the same type.

B A_{IL}, Formally

The formalisation of A_{IL} borrows extensively from that of the single-module version by Patrignani and Clarke [36].

Define $(P_1) \frown (P_2)$ as follows.

$$\begin{array}{c}
\text{(program compatibility)} \\
\hline
\begin{array}{l}
t_1 = \overline{EM}_1; \overline{EO}_1; \overline{RM}_1; \overline{RO}_1 \quad t_2 = \overline{EM}_2; \overline{EO}_2; \overline{RM}_2; \overline{RO}_2 \\
\overline{EM}_1 \Vdash \overline{RM}_2 \quad \overline{EM}_2 \Vdash \overline{RM}_1 \quad \overline{EO}_1 \Vdash \overline{RO}_2 \quad \overline{EO}_2 \Vdash \overline{RO}_1
\end{array} \\
\hline
(m_1, \overline{s}_1, t_1) \frown (m_2, \overline{s}_2, t_2) \\
\hline
\begin{array}{ll}
\text{(Exported satisfy required -meth- ind)} & \text{(Exported satisfy required -meth- base)} \\
\forall i \in 1..n \quad \overline{EM} \Vdash \overline{RM}_i & \exists a. \quad m : M_t \mapsto a \in \overline{EM} \\
\hline
\overline{EM} \Vdash \overline{RM}_1, \dots, \overline{RM}_n & \overline{EM} \Vdash m : M_t \mapsto \iota; \sigma \\
\text{(Exported satisfy required -obj- ind)} & \text{(Exported satisfy required -obj- base)} \\
\forall i \in 1..n \quad \overline{EO} \Vdash \overline{RO}_i & \exists n. \quad o : c \mapsto n \in \overline{EO} \\
\hline
\overline{EO} \Vdash \overline{RO}_1, \dots, \overline{RO}_n & \overline{EO} \Vdash o : c \mapsto \sigma
\end{array}
\end{array}$$

Define a substitution of a symbol σ for a word w as $\eta = [w/\sigma]$.

Define $P_1 + P_2$ as follows.

$$\begin{array}{c}
\text{(program merging)} \\
\hline
\begin{array}{l}
m_1 \frown m_2 \quad \overline{s}_1 \frown \overline{s}_2 \quad t_1 + t_2 = t; \overline{\eta} \quad m = (m_1 + m_2) \overline{\eta} \\
(m_1, \overline{s}_1, t_1) + (m_2, \overline{s}_2, t_2) = (m, \overline{s}_1 \overline{s}_2, t)
\end{array} \\
\hline
\begin{array}{ll}
\text{(mem - compat)} & \text{(desc - compat - ind)} \\
\text{dom}(m_1) \cap \text{dom}(m_2) = \emptyset & \forall i \in 1..n \quad \overline{s} \frown s_i \\
\hline
m_1 \frown m_2 & \overline{s} \frown s_1, \dots, s_n \\
\text{(desc - compat - base)} & \\
\overline{s} = s_1, \dots, s_n \quad \forall i \in 1..n. \quad s_i \equiv (id_i, n_{ci}, n_i) \quad id_i \neq id & \\
\hline
\overline{s} \frown (id, n_c, n) &
\end{array}
\end{array}$$

$$\begin{array}{c}
\text{(mem sub - ind)} \\
\frac{m = a_1 \mapsto w_1 \bar{\eta}, \dots, a_n \mapsto w_n \bar{\eta}}{a_1 \mapsto w_1, \dots, a_n \mapsto w_n \bar{\eta} = m} \\
\\
\text{(sub-s-n)} \quad \frac{n[w/\sigma] = n}{n[w/\sigma] = n} \quad \text{(sub-s-p)} \quad \frac{\pi[w/\sigma] = \pi}{\pi[w/\sigma] = \pi} \quad \text{(sub-s)} \quad \frac{\sigma' = \sigma}{\sigma'[w/\sigma] = w} \\
\\
\text{(sub-s-no)} \quad \frac{\sigma' \neq \sigma}{\sigma'[w/\sigma] = \sigma'} \quad \text{(mem add)} \quad \frac{m_1 \frown m_2}{m_1 + m_2 = m_1 m_2} \\
\\
\text{(linking tables merging)} \\
\frac{\begin{array}{l} \overline{EM}_1; \overline{EO}_1; \overline{RM}_1; \overline{RO}_1 \frown \overline{EM}_2; \overline{EO}_2; \overline{RM}_2; \overline{RO}_2 \\ \overline{EM} = \overline{EM}_1 \overline{EM}_2 \quad \overline{EO} = \overline{EO}_1 \overline{EO}_2 \quad t = \overline{EM}; \overline{EO}; \\ \overline{RM}_1 + \overline{EM}_2 = \overline{RM}' + \bar{\eta}' \quad \overline{RM}_2 + \overline{EM}_1 = \overline{RM}'' + \bar{\eta}'' \\ \overline{RO}_1 + \overline{EO}_2 = \overline{RO}' + \bar{\eta}''' \quad \overline{RO}_2 + \overline{EO}_1 = \overline{RO}'' + \bar{\eta}'''' \\ \overline{RM} = \overline{RM}' \overline{RM}'' \quad \overline{RO} = \overline{RO}' \overline{RO}'' \quad \bar{\eta} = \bar{\eta}' \bar{\eta}'' \bar{\eta}''' \bar{\eta}'''' \end{array}}{\overline{EM}_1; \overline{EO}_1; \overline{RM}_1; \overline{RO}_1 + \overline{EM}_2; \overline{EO}_2; \overline{RM}_2; \overline{RO}_2 = t; \bar{\eta}} \\
\\
\text{(rm - merged - em - ind)} \\
\frac{\begin{array}{l} \overline{RM} = RM_1, \dots, RM_n \quad \forall i \in 1..n \quad RM_i + \overline{EM} = RM'_i + \eta'_i \eta''_i \\ \overline{RM}' = RM'_1, \dots, RM'_n \quad \bar{\eta}' = \eta'_1 \eta''_1, \dots, \eta'_n \eta''_n \end{array}}{\overline{RM} + \overline{EM} = \overline{RM}' + \bar{\eta}'} \\
\\
\text{(rm - merged - em - base)} \\
\frac{m : M_t \mapsto (id, n) \in \overline{EM}}{m : M_t \mapsto \iota; \sigma + \overline{EM} = \emptyset + [n/\sigma][id/\iota]} \\
\\
\text{(rm - merged - em - base2)} \\
\frac{m : M_t \notin \text{dom}(\overline{EM})}{m : M_t \mapsto \iota; \sigma + \overline{EM} = m : M_t \mapsto \iota; \sigma + \emptyset} \\
\\
\text{(ro - merged - eo - ind)} \\
\frac{\begin{array}{l} \overline{RO} = RO_1, \dots, RO_n \quad \forall i \in 1..n \quad RO_i + \overline{EO} = RO'_i + \eta'_i \\ \overline{RO}' = RO'_1, \dots, RO'_n \quad \bar{\eta}' = \eta'_1, \dots, \eta'_n \end{array}}{\overline{RO} + \overline{EO} = \overline{RO}' + \bar{\eta}'} \\
\\
\text{(ro - merged - eo - base)} \\
\frac{m : M_t \mapsto n \in \overline{EM}}{m : M_t \mapsto \sigma + \overline{EM} = \emptyset + [n/\sigma]} \\
\\
\text{(ro - merged - eo - base2)} \\
\frac{m : M_t \notin \text{dom}(\overline{EM})}{m : M_t \mapsto \sigma + \overline{EM} = m : M_t \mapsto \sigma + \emptyset}
\end{array}$$

These rules extend the single-module *PMA* access control policy to account for multiple modules. Assume $s = (id, n_c, n_t)$.

$$\begin{array}{c}
\text{(Aux-unprotected)} \\
\frac{a = (0, n)}{\bar{s} \vdash \text{unprotected}(a)}
\end{array}$$

$$\begin{array}{c}
\text{(Aux-entrypoint)} \\
\frac{a = (id, n') \quad n' = n_b + m \cdot \mathcal{N}_w \quad m \in \mathbb{N} \quad m < n_t}{s \vdash \text{entryPoint}(a)} \\
\text{(Aux-methodEntryPoint)} \\
\frac{a = (id, n') \quad n' = n_b + m \cdot \mathcal{N}_w \quad n' = n_b + (m) \cdot \mathcal{N}_w \quad m \in 1..n_t - 1}{s \vdash \text{entryPoint}(a)} \\
\text{(Aux-returnEntryPoint)} \\
\frac{a = (id, n') \quad n' = n_b + (n_t - 1) \cdot \mathcal{N}_w}{s \vdash \text{returnEntryPoint}(a)} \\
\begin{array}{cc}
\text{(Aux-code)} & \text{(Aux-data)} \\
\frac{a = (id, n') \quad n_b \leq n' < n_b + n_c}{s \vdash \text{codeRange}(a)} & \frac{a = (id, n') \quad n_b + n_c \leq n'}{s \vdash \text{dataRange}(a)} \\
\text{(Aux-range)} & \text{(Aux-internalJump-module)} \\
\frac{a = (id, n') \quad n' \in \mathbb{N}}{s \vdash \text{range}(a)} & \frac{s \vdash \text{codeRange}((id, n), (id, n'))}{s \vdash \text{internalJump}((id, n), (id, n'))} \\
\text{(Aux-writeAllowed-module)} \\
\frac{s \vdash \text{codeRange}(a) \quad s \vdash \text{dataRange}(a')}{s \vdash \text{writeAllowed}(a, a')} \\
\text{(Aux-readAllowed-module)} \\
\frac{s \vdash \text{codeRange}(a) \quad s \vdash \text{range}(a')}{s \vdash \text{readAllowed}(a, a')} \\
\text{(Aux-crossjump)} \\
\frac{\exists s \in \overline{s}. s \vdash \text{range}(a) \quad \exists s' \in \overline{s}. s' \vdash \text{entryPoint}(a') \quad s \neq s'}{\overline{s} \vdash \text{crossJump}(a, a')} \\
\text{(Aux-internalJump)} \\
\frac{\exists s \in \overline{s}. s \vdash \text{internalJump}(a, a')}{\overline{s} \vdash \text{internalJump}(a, a')} \\
\begin{array}{cc}
\text{(Aux-internalJump2)} & \text{(Aux-validJump)} \\
\frac{\overline{s} \vdash \text{unprotected}(a) \quad \overline{s} \vdash \text{unprotected}(a')}{\overline{s} \vdash \text{internalJump}(a, a')} & \frac{\overline{s} \vdash \text{internalJump}(a, a')}{\overline{s} \vdash \text{validJump}(a, a')} \\
\text{(Aux-validJump2)} & \text{(Aux-validJump3)} \\
\frac{\overline{s} \vdash \text{crossJump}(a, a')}{\overline{s} \vdash \text{validJump}(a, a')} & \frac{\overline{s} \vdash \text{unprotected}(a) \quad \overline{s} \vdash \text{entryPoint}(a')}{\overline{s} \vdash \text{validJump}(a, a')} \\
\text{(Aux-validJump4)} & \text{(Aux-writeAllowed)} \\
\frac{\overline{s} \vdash \text{protected}(a) \quad \overline{s} \vdash \text{unprotected}(a')}{\overline{s} \vdash \text{validJump}(a, a')} & \frac{\exists s \in \overline{s}. s \vdash \text{writeAllowed}(a, a')}{\overline{s} \vdash \text{writeAllowed}(a, a')} \\
\text{(Aux-writeAllowed2)} \\
\frac{\overline{s} \vdash \text{unprotected}(a) \quad \overline{s} \vdash \text{unprotected}(a')}{\overline{s} \vdash \text{writeAllowed}(a, a')} \\
\text{(Aux-writeAllowed3)} \\
\frac{\exists s \in \overline{s}. s \vdash \text{range}(a) \quad \overline{s} \vdash \text{unprotected}(a')}{\overline{s} \vdash \text{writeAllowed}(a, a')} \\
\text{(Aux-readAllowed)} \\
\frac{\exists s \in \overline{s}. s \vdash \text{readAllowed}(a, a')}{\overline{s} \vdash \text{readAllowed}(a, a')}
\end{array}
\end{array}$$

$$\begin{array}{c}
\text{(Aux-readAllowed2)} \\
\frac{\overline{s} \vdash \text{unprotected}(a) \quad \overline{s} \vdash \text{unprotected}(a')}{\overline{s} \vdash \text{readAllowed}(a, a')} \\
\text{(Aux-readAllowed3)} \\
\frac{\exists s \in \overline{s}. s \vdash \text{range}(a) \quad \overline{s} \vdash \text{unprotected}(a')}{\overline{s} \vdash \text{readAllowed}(a, a')} \\
\text{(Aux-readAllowed4)} \\
\frac{\overline{s} \vdash \text{unprotected}(a) \quad \exists s \in \overline{s}. s \vdash \text{entryPoint}(a')}{\overline{s} \vdash \text{readAllowed}(a, a')} \\
\begin{array}{cc}
\text{(Aux-methodEntryPoint)} & \text{(Aux-entryPoint)} \\
\frac{\exists s \in \overline{s}. s \vdash \text{methodEntryPoint}(a')}{\overline{s} \vdash \text{methodEntryPoint}(a)} & \frac{\exists s \in \overline{s}. s \vdash \text{entryPoint}(a')}{\overline{s} \vdash \text{entryPoint}(a)} \\
\text{(Aux-returnEntryPoint)} & \text{(Aux-protected)} \\
\frac{\exists s \in \overline{s}. s \vdash \text{returnEntryPoint}(a')}{\overline{s} \vdash \text{returnEntryPoint}(a)} & \frac{\exists s \in \overline{s}. s \vdash \text{range}(a')}{\overline{s} \vdash \text{protected}(a)}
\end{array} \\
\text{(Aux-currentModule)} \\
\frac{s \in \overline{s} \quad s = id, n_c, n_t \quad s \vdash \text{range}(id, n)}{(id, n) \vdash \text{currentModule}(\overline{s}, s)} \\
\text{(Aux-exitJump)} \\
\frac{\exists s \in \overline{s}. s \vdash \text{codeRange}(a) \quad t = \overline{EM}; \overline{EO}; \overline{RM}; \overline{RO} \quad m : t(\overline{l}) \rightarrow t \mapsto w, w' \in \overline{RM}}{\overline{s}, t \vdash \text{exitJump}(a, w, w')} \\
\begin{array}{cc}
\text{(Aux-stuck)} & \text{(Aux-validJump-traces)} \\
\frac{m(p) = w \quad w \notin \mathcal{I}}{\vdash \text{stuck}(p', r', f', m', \overline{s}, h)} & \frac{}{\overline{s} \vdash \text{validJump}(p, a)} \\
\text{(Aux-validJump-traces2)} & \text{(Aux-forwardReturn-EP)} \\
\frac{}{\overline{s}, t \vdash \text{exitJump}(p, a)} & \frac{p \equiv (1, 3 * \mathcal{N}_w)}{\overline{s}, t \vdash \text{forwardReturnEP}(p)}
\end{array}
\end{array}$$

An exit jump is only used for the trace semantics, so where states span just some modules and the descriptors describe just their layout. An exit jump is therefore a jump to an address that is not in the domain covered by the modules. This could be completely another module or the unprotected code.

The rules presented in the paper extend the operational semantics of AIL. Define a module state as follows: $\Upsilon ::= (p, r, f, m, s, h)$ and a program state as follows: $\Omega ::= (p, r, f, m, \overline{s}, h)$. Let Π denote the set of all symbolic nonces π . Let \equiv denote syntactic equivalence; equating a number and a symbol therefore results in false.

$$\begin{array}{c}
\text{(Eval-module-common)} \\
\frac{(p, r, f, m, s, h) \xrightarrow{id} (p', r', f', m', s, h') \quad s = (id, n_c, n_t) \quad p = (id, n) \quad s \vdash \text{internalJump}(p, p')}{(p, r, f, m, s, h) \xrightarrow{id} (p', r', f', m', s, h')} \\
\text{(Eval-movi)} \\
\frac{p = (id, n) \quad m(p) = (\text{movi } r_a \ i) \quad r' = r[r_a \mapsto i]}{(p, r, f, m, s, h) \xrightarrow{id} ((id, n+1), r', f, m, s, h)}
\end{array}$$

$$\begin{array}{c}
\text{(Eval-add)} \\
\frac{
\begin{array}{l}
p = (id, n) \quad m(p) = (\text{add } r_d \ r_s) \\
v_d = r(r_d), \text{ if } r(r_d) \in \Pi \text{ then } v_d = 0 \\
v_s = r(r_s), \text{ if } r(r_s) \in \Pi \text{ then } v_s = 0 \\
v = v_d + v_s \quad r' = r[r_d \mapsto v] \quad f' = f[\text{ZF} \mapsto (v == 0)]
\end{array}
}{
(p, r, f, m, s, h) \xrightarrow{id} ((id, n+1), r', f', m, s, h)
} \\
\text{(Eval-sub)} \\
\frac{
\begin{array}{l}
p = (id, n) \quad m(p) = (\text{sub } r_d \ r_s) \quad v_d = r(r_d), \text{ if } r(r_d) \in \Pi \text{ then } v_d = 0 \\
v_s = r(r_s), \text{ if } r(r_s) \in \Pi \text{ then } v_s = 0 \quad v = v_d - v_s \quad r' = r[r_d \mapsto \text{abs}(v)] \\
f' = f[\text{ZF} \mapsto (v == 0); \text{SF} \mapsto (v < 0)]
\end{array}
}{
(p, r, f, m, s, h) \xrightarrow{id} ((id, n+1), r', f', m, s, h)
} \\
\text{(Eval-cmp)} \\
\frac{
p = (id, n) \quad m(p) = (\text{cmp } r_d \ r_s) \quad f' = f[\text{ZF} \mapsto (r(r_d) \equiv r(r_s))]
}{
(p, r, f, m, s, h) \xrightarrow{id} ((id, n+1), r, f', m, s, h)
} \\
\text{(Eval-je-true)} \\
\frac{
p = (id, n) \quad m(p) = (\text{je } r_d \ f_i) \quad n' = r(r_d) \quad f(f_i) = 1
}{
(p, r, f, m, s, h) \xrightarrow{id} ((id, n'), r, f, m, s, h)
} \\
\text{(Eval-je-false)} \\
\frac{
p = (id, n) \quad m(p) = (\text{je } r_d \ f_i) \quad f(f_i) = 0
}{
(p, r, f, m, s, h) \xrightarrow{id} ((id, n+1), r, f, m, s, h)
} \\
\text{(Eval-new)} \\
\frac{
p = (id, n) \quad m(p) = \text{new } r_d \quad r' = r[r_d \mapsto \pi] \quad \pi \notin h
}{
(p, r, f, m, s, h) \xrightarrow{id} ((id, n+1), r', f, m, s, h; \pi)
} \\
\text{(Eval-zero)} \\
\frac{
p = (id, n) \quad m(p) = (\text{zero}) \quad \forall i \in \mathbb{N} \quad r' = r[r_i \mapsto 0]
}{
(p, r, f, m, s, h) \xrightarrow{id} ((id, n+1), r', f, m, s, h)
} \\
\text{(Eval-halt)} \\
\frac{
p = (id, n) \quad m(p) = (\text{halt})
}{
(p, r, f, m, s, h) \xrightarrow{id} ((0, -1), r, f, m, s, h)
} \\
\text{(Eval-single-to-multiple-modules)} \\
\frac{
(p, r, f, m, s, h) \xrightarrow{id} (p', r', f', m', s, h') \quad p \vdash \text{currentModule}(\overline{s}, s) \quad p = (id, n)
}{
(p, r, f, m, \overline{s}, h) \rightarrow (p', r', f', m', \overline{s}, h')
} \\
\text{(Eval-movl)} \\
\frac{
\begin{array}{l}
p = (id, n) \quad m(p) = (\text{movl } r_d \ r_s \ r_i) \\
\overline{s} \vdash \text{readAllowed}(n, (r(r_s), r(r_i))) \quad r' = r[r_d \mapsto m(r(r_s), r(r_i))]
\end{array}
}{
(p, r, f, m, \overline{s}, h) \rightarrow ((id, n+1), r', f, m, \overline{s}, h)
} \\
\text{(Eval-movs)} \\
\frac{
\begin{array}{l}
p = (id, n) \quad m(p) = (\text{movs } r_d \ r_s \ r_i) \\
\overline{s} \vdash \text{writeAllowed}(n, (r(r_d), r(r_i))) \quad m' = m[(r(r_d), r(r_i)) \mapsto r(r_s)]
\end{array}
}{
(p, r, f, m, \overline{s}, h) \rightarrow ((id, n+1), r, f, m', \overline{s}, h)
} \\
\text{(Eval-jmp)} \\
\frac{
\begin{array}{l}
p = (id, n) \quad m(p) = (\text{jmp } r_d \ r_i) \quad n' = r(r_d) \quad id' = r(r_i) \quad r' = r[r_0 \mapsto id] \\
\overline{s} \vdash \text{validJump}(p, (id', n'))
\end{array}
}{
(p, r, f, m, \overline{s}, h) \rightarrow ((id', n'), r', f, m, \overline{s}, h)
}
\end{array}$$

B.0.1 Trace Equivalence for Securely-Compiled Components in AIL

This section defines a trace equivalence to reason about securely-compiled components in AIL; it is inspired by the trace semantics for the single-module version of AIL [36].

Trace equivalence is a simpler tool to reason about equivalence of components than contextual equivalence [36, 35, 4, 24, 25]. Trace equivalence relates two components that exhibit the same trace semantics. Trace semantics describes the behaviour of a component as a set of traces. Traces are sequences of actions $\bar{\alpha}$ with the following syntax:

$$\begin{array}{ll} \text{labels} & \lambda ::= \alpha \mid \tau \\ \text{observables} & \gamma ::= \text{call } a \bar{w} \mid \text{ret } a w, id \end{array} \quad \begin{array}{ll} \text{actions} & \alpha ::= \gamma! \mid \gamma? \mid \checkmark \end{array}$$

Labels λ generated by the trace semantics can be actions α or the silent action τ . The silent action τ is generated by unobservable transitions, thus these labels are not accumulated in traces. Actions α can be an observable, decorated action γ or a tick \checkmark , which indicates termination of the computation. Observable actions γ are either a function call to an address a with parameters \bar{w} , or a return of value w to address a from the module with id id . Decorations $?$ or $!$ indicate whether unprotected code performs the action ($?$) or receives it ($!$).

Define a (compiled) component state with Θ , it indicates either that the execution is within a component or that the execution is in an unknown location. Formally: $\Theta ::= (p, r, f, m, \bar{s}, h, t) \mid (\text{unk}, m, \bar{s}, h, t)$. $\Theta_0(P, h)$ denotes the initial state of a component $P = (m, \bar{s}, t)$ with a nonce oracle h . Formally: $\Theta_0(P, h) = (\text{unk}, m, \bar{s}, h, t)$. Relation $\xrightarrow{\lambda} \subseteq \Theta \times \lambda \times \Theta$ captures how single labels are generated. The reflexive-transitive closure of $\xrightarrow{\lambda}$ accumulates labels in traces and filters out silent traces; it is captured by relation $\xRightarrow{\bar{\alpha}} \subseteq \Theta \times \bar{\alpha} \times \Theta$. The auxiliary functions adopted for the trace semantics are used to tell if an address is an entry point (`methodEntryPoint`), or if it is the entry point related to the `forwardReturn`(\cdot) procedure in $\llbracket Sys \rrbracket_{\text{AIL}}^{\text{JEM}}$ (`forwardReturnEP`). Function `exitJump` detects jumps outside the memory space of a component; function `stuck` tells if a state is stuck.

Define the trace semantics of a component P as: $\text{Traces}_{\text{AIL}}(P) = \{\bar{\alpha} \mid \forall h. \exists \Theta. \Theta_0(P, h) \xRightarrow{\bar{\alpha}} \Theta\}$. Two modules P_1 and P_2 are trace equivalent, denoted with $P_1 \underline{T} P_2$ if their trace semantics coincides.

Definition 4 (Trace equivalence). $P_1 \underline{T} P_2 \triangleq \text{Traces}_{\text{AIL}}(P_1) = \text{Traces}_{\text{AIL}}(P_2)$.

For securely-compiled components, trace semantics coincides with contextual equivalence (Assumption 5).

Proof Sketch. This proof is analogous to the one for the single-module version of AIL [36], we just give an informal argument why it holds, though the proof of [36] can be easily adapted to scale for AIL. Informally, we need to prove that labels capture all the information that is communicated between a securely-compiled component and external code.

$$\begin{array}{c}
\text{(Trace-call)} \\
\frac{\overline{w} = w_0, \dots, w_{6+k} \quad r = [\mathbf{r}_0 \mapsto w_0, \dots, \mathbf{r}_{6+k} \mapsto w_{6+k}]}{\overline{s} \vdash \text{methodEntryPoint}(p) \quad m : t(t_1 \dots t_k) \rightarrow t \mapsto p \in \overline{EM} \quad t = \overline{EM}; \overline{EO}; \overline{RM}; \overline{RO}} \\
\hline
(\text{unk}, m, \overline{s}, h, t) \xrightarrow{\text{call } p \ \overline{w}^?} (p, r, f, m, \overline{s}, h, t) \\
\text{(Trace-outcall)} \\
\frac{m(p) = (\text{jmp } \mathbf{r}_d \ \mathbf{r}_i) \quad p' = (\iota, \sigma) \quad r(\mathbf{r}_d) \mapsto \sigma \quad r(\mathbf{r}_i) \mapsto \iota \quad \overline{w} = w_0, \dots, w_{6+k} \quad r = [\mathbf{r}_0 \mapsto w_0, \dots, \mathbf{r}_{6+k} \mapsto w_{6+k}]}{\overline{s}, t \vdash \text{exitJump}(p, p') \quad r(\mathbf{r}_5) = 3 * \mathcal{N}_w \quad m : t(t_1 \dots t_k) \rightarrow t \mapsto \iota; \sigma \in \overline{RM} \quad t = \overline{EM}; \overline{EO}; \overline{RM}; \overline{RO}} \\
\hline
(p, r, f, m, \overline{s}, h, t) \xrightarrow{\text{call } p' \ \overline{w}^!} (\text{unk}, m, \overline{s}, h, t) \\
\text{(Trace-returnback)} \\
\frac{r = [\mathbf{r}_0 \mapsto id', \mathbf{r}_6 \mapsto w] \quad \overline{s} \vdash \text{forwardReturnEP}(p)}{(\text{unk}, m, \overline{s}, h, t) \xrightarrow{\text{ret } p \ w, id'^?} (p, r, f, m, \overline{s}, h, t)} \\
\text{(Eval-return)} \\
\frac{m(p) = (\text{jmp } \mathbf{r}_d \ \mathbf{r}_i) \quad p = (id, n) \quad p' = (id', n') \quad r(\mathbf{r}_d) \mapsto n' \quad r(\mathbf{r}_i) \mapsto id' \quad r = [\mathbf{r}_6 \mapsto w] \quad \overline{s}, t \vdash \text{exitJump}(p, p') \quad r(\mathbf{r}_5) \mapsto 0}{(p, r, f, m, \overline{s}, h, t) \xrightarrow{\text{ret } p' \ w, id!} (\text{unk}, m, \overline{s}, h, t)} \\
\hline
\begin{array}{cccc}
\text{(Trace-tau)} & \text{(Trace-refl)} & \text{(Trace-tau)} & \text{(Trace-trans)} \\
\frac{\Theta \rightarrow \Theta'}{\Theta \xrightarrow{\tau} \Theta'} & \frac{}{\Theta \xRightarrow{\epsilon} \Theta} & \frac{\Theta \xrightarrow{\tau} \Theta'}{\Theta \xRightarrow{\epsilon} \Theta'} & \frac{\Theta \xrightarrow{\alpha} \Theta'' \quad \Theta'' \xRightarrow{\overline{\alpha}'} \Theta'}{\Theta \xRightarrow{\overline{\alpha} \cdot \overline{\alpha}'} \Theta'} \\
\end{array} \\
\text{(Trace-termination)} \\
\frac{\Theta \rightarrow \Theta' \quad \vdash \text{stuck}(\Theta')}{\Theta \xrightarrow{\checkmark} \Theta'}
\end{array}$$

!-decorated actions . The **ret** $\dots!$ action has a standard structure for when it is valid: **ret** $a\ w, 1!$, where a has been received on r_0 and r_5 beforehand by $\llbracket Sys \rrbracket_{AIL}^{JEM}$.

Also **call** $\dots!$ has a standard structure: **call** $id, n\ (1, n, 0, 0, 0, 3*\mathcal{N}_w, \overline{w})!$, where id, n is in the export tables of the modules.

Assumption 4 about $\langle \cdot \rangle$ code ensures that all communication coming from a compiled component happen only via entry points.

Specifically, this communication is regulated by the code added via the **protect**(\cdot) function.

In case of a **call** $\dots!$ only the registers used to carry a parameter (including the current object) are passed because **resetFlags**(\cdot), **resetRegisters**(\cdot) and **resetRegistersExcept**(\cdot) ensure that other registers are always 0.

In case of a **ret** $\dots!$ **resetFlags**(\cdot), **resetRegisters**(\cdot) and **resetRegistersExcept**(\cdot) ensure that only the module id and the returned value (in r_6) are not always 0.

Some of the data communicated via registers could be omitted since, as stated, it is always the same so it conveys no information. We keep this data as we chose for the same structure of labels for incoming and outgoing actions.

?-decorated actions . Nor the code inserted by **protect**(\cdot) nor the one generated by $\langle \cdot \rangle$ uses flags as set by external code, so flags are never used to convey information.

In case of a **call** $\dots?$, due to the assumption on the correctness of $\langle \cdot \rangle$ and on the definition of **protect**(\cdot) only the registers used to carry parameters are used for computation.

In case of a **ret** $\dots?$, the only registers used for calculation by **protect**(\cdot) are r_0 and r_6 , which are the only ones captured by the trace semantics. \square

C Algorithm Formalisation

C.1 Trace Back-Translation Algorithm $\langle \cdot \rangle$

Informally, the algorithm is used to build a JEM context that can distinguish two JEM components whose compiled counterparts are trace inequivalent.

The algorithm inputs two distinct AIL traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ and the two JEM components that generate them \mathcal{C}_1 and \mathcal{C}_2 . In other words $\overline{\alpha_1} \in \text{Traces}_{AIL}(\llbracket \mathcal{C}_1 \rrbracket_{AIL}^{JEM})$ and $\overline{\alpha_2} \in \text{Traces}_{AIL}(\llbracket \mathcal{C}_2 \rrbracket_{AIL}^{JEM})$. The algorithm outputs a single JEM component \mathcal{C} that is the context that can differentiate between \mathcal{C}_1 and \mathcal{C}_2 . Formally: $\langle \mathcal{C}_1, \mathcal{C}_2, \overline{\alpha_1}, \overline{\alpha_2} \rangle = \mathcal{C}$. The two traces are the same up to the last action, therefore $\overline{\alpha_1} \equiv \overline{\alpha_1}!$ and $\overline{\alpha_2} \equiv \overline{\alpha_2}!$ where $\alpha_1! \neq \alpha_2!$. Trace $\overline{\alpha}$ is called the *common*

prefix while the two actions $\alpha_1!$ and $\alpha_2!$ are the *different actions*, which appear at index i .

Intuitively, the algorithm produces code \mathbb{C} that replicates all $?$ -decorated actions in the common prefix $\overline{\alpha}$. The $!$ -decorated actions are made by the component that fills the hole of \mathbb{C} and the algorithm adds code to \mathbb{C} to update its internal state. Then, \mathbb{C} performs the differentiation by terminating in case it detects \mathcal{C}_1 and diverging in the other.

The algorithm is divided in three sub-routines: building the skeleton ($\text{skeleton}(\mathcal{C}_1, \mathcal{C}_2)$), emulating the common prefix ($\text{emulate}(\overline{\alpha}, t)$) and distinguishing the different actions ($\text{diff}(\alpha_1!, \alpha_2!, i)$).

$\text{skeleton}(\mathcal{C}_1, \mathcal{C}_2)$ implements all classes and objects that \mathcal{C}_1 and \mathcal{C}_2 specify in their import declarations. Additionally, it creates helper functions and objects, e.g., it has tables where all globally-known objects are stored and it has a variable to keep track of the action being emulated.

$\text{emulate}(\overline{\alpha}, t)$ returns method bodies that fill the classes created by $\text{skeleton}(\cdot)$ based on the different encountered actions. Intuitively, this algorithm subroutine creates a JEM context that emulates all $?$ -actions in the traces.

- **call $a \overline{w}?$** In this case the context must call method m compiled at address a ; t tells which AIL addresses correspond to which JEM methods. Based on the (known) signature of m , the parameters \overline{w} are emulated to their JEM counterpart.

Primitively-typed parameters are emulated to their JEM counterpart. For example when a **Bool** is expected and (true) is received, the parameter is emulated by **true**. Object-typed parameters are stored in globally-accessible tables based on the encoding of their type and of the id they have in AIL. These tables have getters and setters to retrieve objects based on their AIL id.

- **ret $a w?$** In this case the produced code must return the value emulating w , as discussed in the call case.
- **call $a \overline{w}!$ and ret $a w!$** In these cases, the internal state of \mathbb{C} is updated. For example, \mathbb{C} keeps track of the index of the action that is emulating and of all allocated objects. In these cases, the index is incremented by 1 and all newly allocated objects received via \overline{w} or w are added to the tables.

There are cases in which a $?$ -decorated action cannot be emulated, e.g., a return when no method call was made or a call to a method that expects a **Unit**-typed parameter and instead receives (true) . These are actions that try to violate JEM abstractions; in this cases the algorithm must *fail* and generate a differentiating component that does nothing (Lemma 1).

Lemma 1 (Termination is emulation failure). $\forall \mathcal{C}, \langle \mathcal{C} \rangle = (m_c, m_d, t). \overline{\alpha} \checkmark \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C} \rrbracket_{\text{AIL}}^{\text{JEM}}) \iff \text{emulate}(\overline{\alpha}, t) = \text{fail}.$

Lemma 1 is the most important result of this paper, as proving it reveals which issues arise in a secure compiler. This Lemma states that any traces that causes a termination tick cannot be replicated at the source level (i.e., the algorithm *fails*). An action that cannot be replicated in JEM is an action that tries to violate the abstractions of JEM. Termination ticks are only generated by the execution of `halt`; only the checks inserted by `protect(·)` or by $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$ insert that instruction. Traces with a \checkmark are traces that trigger a check; so they are violating a JEM abstraction in compiled code. If this is true for all traces with a \checkmark (the \Rightarrow direction), and also all traces that incur violations generate a \checkmark (the \Leftarrow direction) then the algorithm is correct. Most importantly, this proof indicates where to place the checks at entry points and what to check. The fact that it is not possible to emulate something (e.g., a `Bool`-typed parameter that is not $\langle \text{true} \rangle$ nor $\langle \text{false} \rangle$) indicates what checks to add (in this case, that `Bool`-typed parameters must be either $\langle \text{true} \rangle$ or $\langle \text{false} \rangle$, i.e., function `dynamicTypechecks(·)`).

`diff($\alpha_1!$, $\alpha_2!$, i)` returns two different code fragments that will detect whether the produced context is interacting with \mathcal{C}_1 or \mathcal{C}_2 . For example, consider the two actions to be `ret a $\langle \text{true} \rangle!$` and `ret a $\langle \text{false} \rangle!$` . After the last emulated action, the context needs to check the returned value and terminate in case it is `true` and diverge in the other.

All cases when two `!`-decorated actions are different are considered in `diff(·)`; they are omitted for space reasons.

C.2 The Algorithm, Formally $\langle\langle \cdot \rangle\rangle$

`skeleton($\mathcal{C}_1, \mathcal{C}_2$)` returns a component \mathcal{C} .

`emulate($\overline{\alpha}, t$)` returns a list of expressions bound to the method where they must be added.

`diff($\alpha_1, \alpha_2, i, M, V, \overline{m}, \overline{t}$)` returns a pair of expressions bound to the methods where they must be added.

Define code additions A as follows: $A ::= E@m \mid E@m@i$.

$\mathcal{C} + E@m$ returns a new component where the body of method m has been extended with E .

$\mathcal{C} + E@m@i$ is analogous but the extension E is placed in the body of method m inside an `if` statement whose guard is `oc.isStep(i)`. The necessity of this will be clear after having seen the rest of the functions.

The algorithm can be defined as follows:

$$\begin{array}{c}
\langle\langle\cdot\rangle\rangle \\
\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} = (m_1, \bar{s}, t) \quad \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}} = (m_2, \bar{s}, t) \\
\text{skeleton}(\mathcal{C}_1, \mathcal{C}_2) = \mathcal{C} \\
\text{emulate}(\bar{\alpha}, t) = \bar{A}, M, V, \bar{m}, \bar{t}, i \\
\text{diff}(\alpha_1, \alpha_2, i, M, V, \bar{m}, \bar{t}) = A_1, A_2 \\
\hline
\langle\langle \mathcal{C}_1, \mathcal{C}_2, \bar{\alpha}\alpha_1, \bar{\alpha}\alpha_2 \rangle\rangle = \mathcal{C} + \bar{A} + A_1 + A_2
\end{array}$$

C.3 skeleton($\mathcal{C}_1, \mathcal{C}_2$)

$$\begin{array}{c}
\text{(Collect Interfaces - base)} \\
\bar{M} = \text{public } m_1(\bar{x}_1) : M_{t_1} \{ \text{return } E_1; \}, \dots, \text{public } m_k(\bar{x}_k) : M_{t_k} \{ \text{return } E_k; \} \\
I = \text{class-decl } c\{M_{t_1}, \dots, M_{t_k}\} \\
\hline
\text{interfaces}(\text{import } \bar{I}; \bar{X}; \text{class } c\{K \bar{F}_t \bar{M}\}; \bar{O}) = I \\
\text{(Collect Interfaces - inductive)} \\
\forall i \in 1..k \quad \text{interfaces}(C_i) = I_i \\
\hline
\text{interfaces}(C_1, \dots, C_k) = I_1, \dots, I_k \\
\text{(Collect externs - base)} \\
\bar{O} = \text{object } o_1 : c\{\bar{F}_1\}, \dots, \text{object } o_k : c\{\bar{F}_k\} \\
\bar{X}' = \text{obj-decl } o_1 : c, \dots, \text{obj-decl } o_k : c; \\
\hline
\text{externs}(\text{import } \bar{I}; \bar{X}; \text{class } c\{K \bar{F}_t \bar{M}\}; \bar{O}) = \bar{X}' \\
\text{(Collect externs - inductive)} \\
\forall i \in 1..k \quad \text{externs}(C_i) = X_i \\
\hline
\text{externs}(C_1, \dots, C_k) = \bar{X}_1, \dots, \bar{X}_k \\
\text{(stub-class)} \\
C = \text{import } \bar{I}; \bar{X}; \text{class } i\{K \emptyset \bar{M}\}; \bar{O}; O \quad K = i(\emptyset) \{ \} \\
\text{staticFor}(i) = O \quad i \mapsto \bar{O} \in i \mapsto \bar{O} \\
M_t = M_{t_1}, \dots, M_{t_k} \quad \forall i \in 1..k. \quad \text{stub-method}(M_{t_i}) = M_i \\
\bar{M} = M_1, \dots, M_k \\
\hline
\text{stub-class}(\text{class-decl } i\{\bar{M}_t\}, \bar{I}; \bar{X}, i \mapsto \bar{O}) = C \\
\text{(stub-class inductive)} \\
\forall i \in 1..k \quad \text{stub-class}(I_i) = C_i \\
\hline
\text{stub-class}(I_1, \dots, I_k, \bar{I}; \bar{X}, i \mapsto \bar{O}) = C_1, \dots, C_k \\
\text{(stub-obj)} \\
\text{object } o : t\{\emptyset\} = O \\
\hline
\text{stub-obj}(\text{obj-decl } o : t;) = c \mapsto O \\
\text{(stub-obj ind)} \\
\forall i \in 1..k \quad \text{stub-obj}(X_i) = c_i \mapsto O_i \\
\hline
\text{stub-obj}(X_1, \dots, X_k) = c_1 \mapsto O_1 \uplus \dots \uplus c_k \mapsto O_k \\
\text{(stub-method-unit)} \\
M = \text{public } m(\bar{x}) : t(\bar{t}) \rightarrow \text{Unit} \{ \text{return unit}; \} \\
\hline
\text{stub-method}(m : t(\bar{t}) \rightarrow \text{Unit}) = M \\
\text{(stub-method-bool)} \\
M = \text{public } m(\bar{x}) : t(\bar{t}) \rightarrow \text{Bool} \{ \text{return true}; \} \\
\hline
\text{stub-method}(m : t(\bar{t}) \rightarrow \text{Bool}) = M
\end{array}$$

$$\begin{array}{c}
\text{(stub-method-int)} \\
\frac{M = \text{public } m(\overline{x}) : t(\overline{t}) \rightarrow \text{Int} \{ \text{return } 0; \}}{\text{stub} - \text{method}(m : t(\overline{t}) \rightarrow \text{Int}) = M} \\
\text{(stub-method-obj)} \\
\frac{M = \text{public } m(\overline{x}) : t(\overline{t}) \rightarrow c \{ \text{return null}; \}}{\text{stub} - \text{method}(m : t(\overline{t}) \rightarrow c) = M} \\
\text{(staticfor)} \\
\frac{\text{staticFor}(i) = \text{object } \text{static} - \text{for} - i : i \{ \emptyset \}}{\text{(skel)}} \\
\frac{\begin{array}{l} \text{interfaces}(C_1, \dots, C_k) = \overline{I} \quad \text{externs}(C_1, \dots, C_k) = \overline{E} \\ \forall i \in 1..k. \quad C_i = \text{import } \overline{I}_i; \overline{X}_i; \text{ class } c_i \{ K_i \overline{F}_{t_i} \overline{M}_i \}; \overline{O}_i \\ \overline{I}' = \overline{I}_1, \dots, \overline{I}_k \quad \overline{X}' = \overline{X}_1, \dots, \overline{X}_k \quad \text{helpers}() = C \\ \text{stub} - \text{obj}(\overline{X}') = c \mapsto \overline{O} \quad \text{stub} - \text{class}(\overline{I}', \overline{I}, \overline{X}, c \mapsto \overline{O}) = C \end{array}}{\text{skeleton}(C_1, \dots, C_k, C'_1, \dots, C'_k) = C; C} \\
\text{(helpers)} \\
\frac{\begin{array}{l} C = \text{import } \emptyset; \emptyset; \text{ class } \text{Helper} \{ K \overline{F}_t \overline{M} \}; \overline{O} \\ \text{helper} - \text{fieldtypes}() = \overline{F}_t \\ \text{helper} - \text{objects}() = \overline{O} \quad \text{helper} - \text{methods}() = \overline{M} \\ \text{helper} - \text{constructor}() = K \end{array}}{\text{helpers}() = C} \\
\begin{array}{cc}
\text{(helpers-fieldtypes)} & \text{(helpers-objects)} \\
\frac{\overline{F}_t = \text{step} : \text{Int}}{\text{helper} - \text{fieldtypes}() = \overline{F}_t} & \frac{\text{object } oc : \text{Helper} \{ \text{step} = 0 \}}{\text{helper} - \text{objects}() = O} \\
\text{(helpers-methods)} \\
\begin{array}{l} \text{public } \text{isStep}(x) : M_t \{ \text{return if (this.step == x) \{true\} else \{false\}; } \\ \quad M_t = \text{isStep} : \text{Helper}(\text{Int}) \rightarrow \text{Bool} \\ \text{public } \text{incStep}() : M'_t \{ \text{return this.step} + 1; \text{unit}; \} \\ \quad M'_t = \text{incrStep} : \text{Helper}() \rightarrow \text{Unit} \\ \text{public } \text{diverge}() : M''_t \{ \text{return this.diverge}(); \} \\ \quad M''_t = \text{diverge} : \text{Helper}() \rightarrow \text{Unit} \\ \text{public } \text{main}() : M'''_t \{ \text{return } 0; \} \\ \quad M'''_t = \text{main} : \text{Helper}() \rightarrow \text{Int} \end{array} \\
\hline
\text{helper} - \text{methods}() = \overline{M} \\
\text{(helpers-constructor)} \\
\frac{K = \text{Helper}(\text{step} : \text{Int}) \{ \text{this.step} = 0 \}}{\text{helper} - \text{constructor}() = K}
\end{array}$$

Note that the two components input by `skeleton()` must have the same imports and they must define the same classes, otherwise they are trivial to differentiate. Therefore, to build the skeleton in Rule `skel` we use only one component. Assume the names `Helper` and `oc` are fresh (a simple substitution can ensure this).

For all class `c` and all interface `i` defined in C_1 and C_2 , the `Helper` class also contains a list. Call this class `listof-t` for a class or interface type `t`. Each element of the list has a `t` field that points to an object of type `t` and an `Int` field that contains the `AN` encoding of object's id. The list implements method `getByName(n)` that inputs a name `n` and returns the object with that name in the list. Additionally, each list has an `append` method. `Helper` also has a

method `createNew-t(n)` that creates a new object of type `t` and adds it to the list `listof-t` with name `n`; this method returns the new object. These lists are populated with all known static objects (i.e., the objects declared in \mathcal{C}_1 and \mathcal{C}_2).

Helper also has generic method `addObject(o, n)` to insert object `o` with name `n` will call `instanceof(o, c)` for all possible class and interface types implemented by \mathcal{C}_1 and \mathcal{C}_2 . Based on the type `c`, the method calls the `append` on list `listof-c`.

Helper also has generic method `getByName(n)` will call `getByName(n)` on all lists until it finds the object to return.

C.4 emulate($\bar{\alpha}, t$)

The `emulate(·)` function *fails* any time any of its sub-parts *fails*. If this happens, it returns an empty method body for the `main` method.

Hoe to read the `emulate(·)` judgment: `emulate($\alpha, M_0, V, \bar{id}, \bar{m}, \bar{t}, i, t$) = $E@m, V', \bar{id}', i', \bar{m}', \bar{t}'$` under method environment M_0 , under knowledge of the allocated objects V , with the stack of caller ids \bar{id} , with the stack of called methods \bar{m} and the stack of return types \bar{t} , at step i , with bindings t , action α produces code E to be placed inside method m , it updates the knowledge of objects with V' , it increments the step to i' , it updates the id stack to \bar{id}' , it updates the methods stack to \bar{m}' and the expected return types to \bar{t}' .

The algorithm uses M to store method bindings, so $M ::= \overline{a \mapsto m : M_t}$ and V to store object bindings, so $V ::= \overline{w : t}$.

$$\frac{\begin{array}{c} \text{(methodKnowledge)} \\ M = \overline{a_1 \mapsto m_1 : M_{t_1}, \dots, a_n \mapsto m_n : M_{t_n}} \end{array}}{\text{methodKnowledge}(\overline{m_1 : M_{t_1} \mapsto a_1}, \dots, \overline{m_n : M_{t_n} \mapsto a_n}) = M} \quad \frac{\begin{array}{c} \text{(objectsKnowledge)} \\ V = \overline{w_1 : t_1, \dots, w_n : t_n} \end{array}}{\text{objectsKnowledge}(\overline{o_1 : t_1 \mapsto w_1}, \dots, \overline{o_n : t_n \mapsto w_n}) = V}$$

`emulate(·)` uses a helper function `nonce – to – int(t)` that translates nonces to integers.

Following is the definition of `emulate(·)`.

$$\frac{\begin{array}{c} \text{(Emulate)} \\ t = \overline{EM; EO; RM; RO} \quad \text{methodKnowledge}(\overline{EM, RM}) = M_0 \\ \text{objectsKnowledge}(\overline{EO, RO}) = V_0 \quad m_0 = \text{main} \\ \text{emulate}(\alpha_1, M_0, V_0, \emptyset, m_0, \emptyset, 0, t) = A_1, V_1, \bar{id}_1, i_1, \bar{m}_1, \bar{t}_1 \\ \forall h \in 2..n, \quad \text{emulate}(\alpha_h, M_0, V_0 + V_{h-1}, \bar{id}_{h-1}, \bar{m}_{h-1}, \bar{t}_{h-1}, i_{h-1}, t) = A_h, V'_h, \bar{id}_h, i_h, \bar{m}_h, \bar{t}_h \\ V' = V'_1 + \dots + V'_n \end{array}}{\text{emulate}(\alpha_1, \dots, \alpha_n, t) = A_1; \dots; A_n, M_0, V', \bar{m}_n, \bar{t}_n, i_{n-1}} \quad \frac{\begin{array}{c} \text{(Emulate- fail)} \\ \text{a fail happens} \end{array}}{\text{emulate}(\alpha_1, \dots, \alpha_n) = \emptyset @ \text{main}}$$

$$\begin{array}{c}
\text{(Emulate - call)} \\
\frac{
\begin{array}{l}
a' = (1, 2 * \mathcal{N}_w) \quad a = \mathbf{r}_3, \mathbf{r}_4 \\
M(a) = m : t(t_1; \dots; t_m) \rightarrow t' \quad \forall i \in 1..m \\
E = \text{if } (\text{oc.isStep}(i)) \text{ then } \text{oc.incrStep}(); + E_1 + \dots + E_m + E' \\
E' = \text{var } o = E^o; \text{var retvar} = o.m(\text{arg-1}, \dots, \text{arg-m}) \\
\text{emulate}(w_6 : t, V) = E^o; \emptyset \quad \text{emulate}(w_{6+i} : t_i, V) = E_i^v, V'_i \\
E_i = \text{var arg-i} = E_i^v \quad V' = V'_1 + \dots + V'_n
\end{array}
}{
\text{emulate}(\text{call } a' w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = E@m, V', w_0 \overline{id}, i+1, m \overline{m}, t' \overline{t}
} \\
\text{(Emulate - call - fail)} \\
\frac{
(w_0, w_5) \text{ is not an executable address in } t
}{
\text{emulate}(\text{call } a w_1, \dots, v_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}
} \\
\text{(Emulate - call - fail2)} \\
\frac{
a' = (1, 2 * \mathcal{N}_w) \quad a = \mathbf{r}_3, \mathbf{r}_4 \quad a \notin M
}{
\text{emulate}(\text{call } a' w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}
} \\
\text{(Emulate - callback)} \\
\frac{
\begin{array}{l}
M(a) = m : t(t_1; \dots; t_m) \rightarrow t' \quad E = \text{oc.incrStep}(); + E_1 + \dots + E_m \\
V' = V'_1 + \dots + V'_m \quad \text{Let the name of the } i\text{-th parameter be } x_i \\
\forall i \in 1..m. E_i, V'_i = \text{possiblyAdd}(w_{6+i}, t_i, x_i)
\end{array}
}{
\text{emulate}(\text{call } a w_0, \dots, w_n!, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = E@m, V', w_0 \overline{id}, i+1, m \overline{m}, t' \overline{t}
} \\
\text{(possiblyAdd)} \\
\frac{
\begin{array}{l}
\text{isInternal}(t) \quad w \notin \text{dom}(V) \\
E = \text{var arg-i} = \text{addObject}(x, \text{nonce-to-int}(w)) \quad V = w : t
\end{array}
}{
\text{possiblyAdd}(w, t, x) = E, V
} \\
\text{(Emulate - returnback)} \\
\frac{
\begin{array}{l}
E = \text{if } (\text{oc.isStep}(i)) \text{ then } \text{oc.incrStep}(); \text{var ret} = E^v; \text{return ret}; \\
\text{emulate}(w : t, V) = E^v, V' \quad w' = id \quad a = (1, 3 * \mathcal{N}_w)
\end{array}
}{
\text{emulate}(\text{ret } a w, w'?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = E@m, V', \overline{id}, i+1, m \overline{m}, \overline{t}
} \\
\text{(Emulate - returnback - fail)} \\
\frac{
w' \neq id \quad a = (1, 3 * \mathcal{N}_w)
}{
\text{emulate}(\text{ret } a w, w'?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}
} \\
\text{(Emulate - returnback - fail2)} \\
\frac{
a \neq (1, 3 * \mathcal{N}_w)
}{
\text{emulate}(\text{ret } a w, w'?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}
} \\
\text{(Emulate - return)} \\
\frac{
\begin{array}{l}
\text{isInternal}(t) \quad w \notin \text{dom}(V) \quad V' = w : t \\
E = \text{oc.incrStep}(); \text{var arg-r} = \text{addObject}(\text{retvar}, \text{nonce-to-int}(w))
\end{array}
}{
\text{emulate}(\text{ret } a w, w'!, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = E@m, V', i+1, \overline{id}, \overline{m}, \overline{t}
} \\
\text{(Emulate - return2)} \\
\frac{
\neg \text{isInternal}(t) \quad E = \text{oc.incrStep}(); \quad V' = \emptyset
}{
\text{emulate}(\text{ret } a w, w'!, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = E@m, V', i+1, \overline{id}, \overline{m}, \overline{t}
} \\
\text{(Emulate - addresses)} \quad \text{(Emulate - addresses - fail)} \\
\frac{
M(a) = m : M_t \quad a \notin \text{dom}(M)
}{
\text{emulate}(a, M) = m : M_t \quad \text{emulate}(a, M) = \text{fail}
} \\
\text{(Emulate - values - unit)} \\
\frac{
t \equiv \text{Unit} \quad w \equiv \langle \text{unit} \rangle \quad E = \text{unit} \quad V' = \emptyset
}{
\text{emulate}(w : t, V) = E, V'
}
\end{array}$$

$$\begin{array}{c}
\text{(Emulate - values - unit fail)} \\
\frac{t \equiv \text{Unit} \quad w \not\equiv \langle \text{Unit} \rangle}{\text{emulate}(w : t, V) = \text{fail}} \\
\text{(Emulate - values - bool true)} \\
\frac{t \equiv \text{Bool} \quad w \equiv \langle \text{true} \rangle \quad E = \text{true} \quad V' = \emptyset}{\text{emulate}(w : t, V) = E, V'} \\
\text{(Emulate - values - bool false)} \\
\frac{t \equiv \text{Bool} \quad w \equiv \langle \text{false} \rangle \quad E = \text{false} \quad V' = \emptyset}{\text{emulate}(w : t, V) = E, V'} \\
\text{(Emulate - values - bool fail)} \\
\frac{t \equiv \text{Bool} \quad w \not\equiv \langle \text{true} \rangle \quad w \not\equiv \langle \text{false} \rangle}{\text{emulate}(w : t, V) = \text{fail}} \\
\text{(Emulate - values - int)} \\
\frac{t \equiv \text{Int} \quad E = \text{integerFor}(w) \quad V' = \emptyset}{\text{emulate}(w : t, V) = E, V'} \\
\text{(Emulate - values - null)} \\
\frac{t <: \text{Obj} \quad w \equiv \langle \text{null} \rangle \quad E = \text{null} \quad V' = \emptyset}{\text{emulate}(w : t, V) = E, V'} \\
\text{(Emulate - values - external old)} \\
\frac{t <: \text{Obj} \quad \text{isExternal}(t) \quad w \in V \quad E = \text{oc.getByName}(\text{nonce} - \text{to} - \text{int}(w_i)) \quad V' = \emptyset}{\text{emulate}(w : t, V) = E, V'} \\
\text{(Emulate - values - external old - fail)} \\
\frac{t <: \text{Obj} \quad \text{isExternal}(t) \quad w : t' \in V \quad t \not<: t'}{\text{emulate}(w : t, V) = \text{fail}} \\
\text{(Emulate - values - external new)} \\
\frac{t <: \text{Obj} \quad \text{isExternal}(t) \quad n \notin \text{dom}(V) \quad E = \text{oc.createNew-t}(n); \quad V' = n : t}{\text{emulate}(n : t, V) = E, V'} \\
\text{(Emulate - values - intern)} \\
\frac{t <: \text{Obj} \quad \text{isInternal}(t) \quad w \in \text{dom}(V) \quad E = \text{oc.getByName}(\text{nonce} - \text{to} - \text{int}(w)) \quad V' = \emptyset}{\text{emulate}(w : t, V) = E, V'} \\
\text{(Emulate - values - intern - fail)} \\
\frac{t <: \text{Obj} \quad \text{isInternal}(t) \quad w \notin V}{\text{emulate}(w : t, V) = \text{fail}} \\
\text{(Emulate - obj)} \\
\frac{t \equiv \text{Obj} \quad w \in \text{dom}(V) \quad E = \text{oc.getByName}(\text{nonce} - \text{to} - \text{int}(w)) \quad V' = \emptyset}{\text{emulate}(w : t, V) = E, V'} \\
\text{(Emulate - obj -ex)} \\
\frac{t \equiv \text{Obj} \quad n \notin \text{dom}(V) \quad E = \text{oc.createNew-t}'(n); \quad V' = n : t'}{\text{emulate}(n : t, V) = E, V'} \\
\text{If } n \text{ is used again, let it be with type } t' \neq \text{Obj} \text{ otherwise let } t' \text{ be any external type} \\
\text{(Emulate - obj -ex fail)} \quad \text{(Emulate-type - unit)} \\
\frac{t \equiv \text{Obj} \quad \pi \notin \text{dom}(V)}{\text{emulate}(\pi : t, V) = \text{fail}} \quad \frac{w \equiv \langle \text{Unit} \rangle}{\text{emulate}(w) = \text{Unit}}
\end{array}$$

$$\begin{array}{c}
\begin{array}{c}
\text{(Emulate-type - bool)} \\
\hline
w = \llbracket \text{Bool} \rrbracket \\
\hline
\text{emulate}(w) = \text{Bool} \\
\text{(intfor - number)}
\end{array}
\quad
\begin{array}{c}
\text{(Emulate-type - Int)} \\
\hline
w = \llbracket \text{Int} \rrbracket \\
\hline
\text{emulate}(w) = \text{Int} \\
\text{(intfor - nonces)}
\end{array}
\quad
\begin{array}{c}
\text{(Emulate-type - class)} \\
\hline
w = \llbracket c \rrbracket \\
\hline
\text{emulate}(w) = c
\end{array}
\\[10pt]
\begin{array}{c}
\text{integerFor}(n) = n \\
\text{(intfor - symbol)} \\
\hline
\text{integerFor}(\sigma) = \text{fail} \\
\text{(isExternal)} \\
\text{Type } t \text{ is an interface in } \mathcal{C}_1 \text{ or } \mathcal{C}_2 \\
\hline
\text{isExternal}(t) \\
\text{(emulate-paper)} \\
\text{linkof}(\mathcal{C}) = t \\
\hline
\text{emulate}(\overline{a}, \mathcal{C}) = \text{emulate}(\overline{a}, t) \\
\text{(linkof-ind)} \\
\forall i \in 1..n. \quad \text{linkof}(C_i) = t_i \\
\hline
\text{linkof}(C_1, \dots, C_n) = t_1 \uplus \dots \uplus t_n
\end{array}
\quad
\begin{array}{c}
\text{integerFor}(\pi) = 0 \\
\text{(isinternal)} \\
\text{Type } t \text{ is a class in } \mathcal{C}_1 \text{ or } \mathcal{C}_2 \\
\hline
\text{isInternal}(t) \\
\text{(isPrimitive)} \\
\text{Type } t \text{ is } \text{Unit}, \text{Bool} \text{ or } \text{Int} \\
\hline
\text{isPrimitive}(t) \\
\text{(linkof-base)} \\
\llbracket C \rrbracket = (m_c, m_d, t) \\
\hline
\text{linkof}(C) = t
\end{array}
\end{array}$$

Since whole `AIL` programs are never executed with free symbols in their memory, Rule `intfor -symbol` is never used.

It is not a problem to accept inputs where the return address is not executable; in those cases there will be an automatic \checkmark . The emulation rules need to keep track of that, as they do with Rule `Emulate - call - fail`.

Emulation of actions to $\llbracket Sys \rrbracket_{AIL}^{JEM}$.

$$\begin{array}{c}
\begin{array}{c}
\text{(Emulate - call - fwcalls - w1)} \\
a \equiv (1, 2 * \mathcal{N}_w) \quad w_0 == id \\
\hline
\text{emulate}(\text{call } a \ w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}
\end{array}
\\[10pt]
\begin{array}{c}
\text{(Emulate - call - fwcalls - w2)} \\
a \equiv (1, 2 * \mathcal{N}_w) \quad w_3 == 1 \\
\hline
\text{emulate}(\text{call } a \ w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}
\end{array}
\\[10pt]
\begin{array}{c}
\text{(Emulate - call - fwret - w1)} \\
a \equiv (1, 3 * \mathcal{N}_w) \quad id \neq w_0 \\
\hline
\text{emulate}(\text{call } a \ w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}
\end{array}
\\[10pt]
\begin{array}{c}
\text{(Emulate - call - fwret - w2)} \\
a \equiv (1, 3 * \mathcal{N}_w) \\
\hline
\text{emulate}(\text{call } a \ w_0, \dots, w_n?, M_0, V, \emptyset, \overline{m}, \overline{t}, i, t) = \text{fail} \\
\text{(Emulate - call - testobj)}
\end{array}
\\[10pt]
\begin{array}{c}
a \equiv (1, 0) \quad t' = \text{Bool} \\
E = \text{if } (\text{oc.isStep}(i)) \text{ then } \text{oc.incrStep}(); \text{var retvar} = \text{instanceof}(E_w, E_t) \\
\text{emulate}(w_7 : E_t, V) = E_w, V' \quad \text{emulate}(w_8) = E_t, \emptyset \\
\hline
\text{emulate}(\text{call } a \ w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = E@m, V', w_0 \overline{id}, i+1, \overline{m}, t' \overline{t} \\
\text{(Emulate - call - testobj - w)} \\
a \equiv (1, 0) \quad w_7 \notin \text{dom}(V) \\
\hline
\text{emulate}(\text{call } a \ w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}
\end{array}
\end{array}$$

$$\begin{array}{c}
\text{(Emulate - call - regobj)} \\
\frac{a \equiv (1, \mathcal{N}_w) \quad t' = \text{Unit} \quad E = \text{if } (\text{oc.isStep}(i)) \text{ then } \text{oc.incrStep}(); \text{ var retvar} = \text{unit};}{\text{emulate}(\text{call } a \ w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = E@m, \emptyset, w_0 \overline{id}, i+1, \overline{m}, t' \overline{t}} \\
\text{(Emulate - call - regobj - w)} \\
\frac{a \equiv (1, \mathcal{N}_w) \quad w_7 \in \text{dom}(V)}{\text{emulate}(\text{call } a \ w_0, \dots, w_n?, M_0, V, \overline{id}, \overline{m}, \overline{t}, i, t) = \text{fail}}
\end{array}$$

C.5 $\text{diff}(\alpha_1, \alpha_2, i, M, V, \overline{m}, \overline{t})$

All cases have their dual, where the actions are flipped. The arguments are: the different action for C_1 , the different action for C_2 , the index preceding those actions, the knowledge of method bindings, the knowledge of object bindings, the stack of current methods and the stack of expected return types. In the following, indicate an empty action with \emptyset .

$$\begin{array}{c}
\text{(Diff-length-tick)} \\
\frac{}{\text{diff}(\checkmark, \emptyset, i, M, V, \overline{m}, \overline{t}) = \emptyset@m@i, \emptyset@m@i} \\
\text{(Diff-length-call)} \\
\frac{E = \text{exit}(1); \quad m : M_t = \text{emulate}(a, M)}{\text{diff}(\text{call } a \ \overline{w}!, \emptyset, i, M, V, \overline{m}, \overline{t}) = E@m@i, E@m@i} \\
\text{(Diff-length-return)} \\
\frac{E = \text{exit}(1);}{\text{diff}(\text{ret } a \ w, w'!, \emptyset, i, M, V, \overline{m}, \overline{t}) = E@m@i, E@m@i} \\
\text{(Diff-rets-primitive)} \\
\frac{\text{isPrimitive}(t) \quad \text{emulate}(w : t, V) = E^r, V'}{E = \text{if } (\text{retvar} == E^r) \{ \text{exit}(1); \} \text{ else } \{ \text{oc.diverge}(); \}} \\
\text{(Diff-rets-addr)} \\
\frac{}{\text{diff}(\text{ret } a \ w, w'!, \text{ret } a \ w_1, w'_1!, i, M, V, \overline{m}, \overline{t}) = E@m@i, E@m@i} \\
\text{the code of } (\cdot) \text{ only uses what was passed in } \mathbf{r}_0 \text{ and } \mathbf{r}_5 \text{ as addresses to return} \\
\text{so this case never arises because the addresses where to return} \\
\text{are always the same (they come from the same ?-decorated actions)} \\
\text{(Diff-rets-internal)} \\
\frac{}{\text{diff}(\text{ret } a' \ w, w'!, \text{ret } a \ w, w'!, i, M, V, \overline{m}, \overline{t}) = \emptyset} \\
\text{(Diff-rets-external)} \\
\frac{\text{isInternal}(t) \quad \text{emulate}(w : t, V) = E^r, V'}{E = \text{if } (\text{retvar} == E^r) \{ \text{exit}(1); \} \text{ else } \{ \text{oc.diverge}(); \}} \\
\text{(Diff-rets-external)} \\
\frac{}{\text{diff}(\text{ret } a \ w, w'!, \text{ret } a \ w_1, w'_1!, i, M, V, \overline{m}, \overline{t}) = E@m@i, E@m@i} \\
\text{(Diff-calls-methods)} \\
\frac{\begin{array}{l} E_1 = \text{exit}(1); \quad \text{emulate}(a, M) = m_1 : M_{t1} \\ E_2 = \text{oc.diverge}(); \quad \text{emulate}(a', M) = m_2 : M_{t2} \end{array}}{\text{diff}(\text{call } a \ \overline{w}!, \text{call } a' \ \overline{w}!, i, M, V, \overline{m}, \overline{t}) = E_1@m_1@i, E_2@m_2@i}
\end{array}$$

$$\begin{array}{c}
\text{(Diff-callee)} \\
\frac{\overline{w} = w_1, \dots, w_n \quad \overline{w}' = w'_1, \dots, w'_n \quad w_5 \neq w'_5 \quad \text{emulate}(a, M) = m : t(t_1, \dots, t_n) \rightarrow t' \quad \text{emulate}(w_5 : t, V) = E^r, V' \quad E = \text{if } (\text{this} == E^r) \{ \text{exit}(1); \} \text{ else } \{ \text{oc.diverge}(); \}}{\text{diff}(\text{call } a \overline{w}!, \text{call } a \overline{w}'!, i, M, V, \overline{m}, \overline{t}) = E@m@i, E@m@i} \\
\text{(Diff-calls2-parameters - primitive)} \\
\frac{\overline{w} = w_1, \dots, w_n \quad \overline{w}' = w'_1, \dots, w'_n \quad \exists i \in 6..n. w_i \neq w'_i \quad \text{emulate}(a, M) = m : t(t_1, \dots, t_n) \rightarrow t' \quad \text{isPrimitive}(t_i) \quad \text{emulate}(w_i : t_i, V) = E^r, V' \quad E = \text{if } (\text{retvar} == E^r) \{ \text{exit}(1); \} \text{ else } \{ \text{oc.diverge}(); \}}{\text{diff}(\text{call } a \overline{w}!, \text{call } a \overline{w}'!, i) = E@m@i, E@m@i} \\
\text{(Diff-calls2-parameters - internal)} \\
\frac{\overline{w} = w_1, \dots, w_n \quad \overline{w}' = w'_1, \dots, w'_n \quad \exists i \in 6..n. w_i \neq w'_i \quad \text{emulate}(a, M) = m : t(t_1, \dots, t_n) \rightarrow t' \quad \text{isInternal}(t_i) \quad \text{emulate}(w_i : t_i, V) = E^r, V' \quad \text{Let the name of the } i\text{-th parameter be } x_i \quad E = \text{if } (x_i == E^r) \{ \text{exit}(1); \} \text{ else } \{ \text{oc.diverge}(); \}}{\text{diff}(\text{call } a \overline{w}!, \text{call } a \overline{w}'!, i) = E@m@i, E@m@i} \\
\text{(Diff-calls2-parameters - external)} \\
\frac{\overline{w} = w_1, \dots, w_n \quad \overline{w}' = w'_1, \dots, w'_n \quad \exists i \in 6..n. w_i \neq w'_i \quad \text{emulate}(a, M) = m : t(t_1, \dots, t_n) \rightarrow t' \quad \text{isExternal}(t_i) \quad \text{Let the name of the } i\text{-th parameter be } x_i \quad \text{emulate}(w_i : t_i, V) = E^r, V' \quad E = \text{if } (x_i == E^r) \{ \text{exit}(1); \} \text{ else } \{ \text{oc.diverge}(); \}}{\text{diff}(\text{call } a \overline{w}!, \text{call } a \overline{w}'!, i) = E@m@i, E@m@i} \\
\text{(Diff-r0-r4)} \\
\frac{\overline{w} = w_1, \dots, w_n \quad \overline{w}' = w'_1, \dots, w'_n \quad \exists i \in 0, 1, 2, 4. w_i \neq w'_i \quad r_0 \text{ to } r_2 \text{ are always reset to } 0 \text{ so this case does not exist} \quad r_4 \text{ contains the returnback entry point address, which is always } 0}{\text{diff}(\text{call } a \overline{w}!, \text{call } a \overline{w}'!, i, M, V, \overline{m}, \overline{t}) = \emptyset} \\
\text{(Diff-acts)} \\
\frac{E_1 = \text{exit}(1); \quad \text{emulate}(a, M) = m_1 : M_t \quad E_2 = \text{oc.diverge}();}{\text{diff}(\text{call } a \overline{w}!, \text{ret } a' w, w'!, i, M, V, m_2 \overline{m}, \overline{t}) = E_1@m_1@i, E_2@m_2@i} \\
\text{(Diff-acts2)} \\
\frac{\text{emulate}(a, M) = m_1 : M_t \quad E_1 = \text{oc.diverge}();}{\text{diff}(\text{call } a \overline{w}!, \sqrt{}, i, M, V, \overline{m}, \overline{t}) = E_1@m_1@i, E_1@m_1@i} \\
\text{(Diff-acts3)} \\
\frac{E_1 = \text{oc.diverge}();}{\text{diff}(\text{ret } a' w, w'!, \sqrt{}, i, M, V, m_1 \overline{m}, \overline{t}) = E_1@m_1@i, E_1@m_1@i}
\end{array}$$

Since the code is placed inside the code generated by the last action i , the code generated by $\text{diff}(\cdot)$ does not need to be wrapped in an $\text{if } (\text{oc.isStep}(i))$.

D Proofs of Section 6

This section contains only proofs and proof sketches that are not found in the paper.

Well-behaved Contextual Equivalence A form of contextual equivalence considering a restricted number of contexts is *well-behaved contextual equivalence*, denoted as $M_1 \simeq_{\text{ctx}}^{\text{w-b}} \text{AIL}_{\text{JEM}} M_2$ [33]. Well-behaved contextual equivalence is defined

for AIL modules with respect to JEM contexts. Well-behaved contextual equivalence is analogous to contextual equivalence except that it considers a subset of AIL contexts, namely those that behave like JEM ones.

The notion of well-behaved contextual equivalence is used for compiler correctness specification, Assumption 4 can be restated as Assumption 6.

Assumption 6 ((\cdot) is correct). $\forall \mathcal{C}_1, \mathcal{C}_2. \mathcal{C}_1 \simeq_{ctx} \mathcal{C}_2 \iff (\mathcal{C}_1) \stackrel{w-b}{\simeq}_{ctx}^{AIL} (\mathcal{C}_2)$.

Proof sketch of Theorem 1

Proof Sketch. The **protect**(\cdot) phase of $\llbracket \cdot \rrbracket_{AIL}^{JEM}$ does not modify how expressions are compiled (i.e., the workings of (\cdot)) except for how **instanceof** is compiled and how external functions are invoked.

Additionally, the methodEP code is prepended to all method calls and the returnEP code is prepended to all returns.

These procedures contain checks that can abort, though correct code never trigger them.

As presented in Problem 5, **instanceof** is compiled to **testObj**(\cdot).

We can see that this is correct by simply inspecting its code.

The other modification done to (\cdot) code is in the way external functions are called.

The chaining of **preamble**(\cdot) ensures that securely-compiled modules are exited via $\llbracket Sys \rrbracket_{AIL}^{JEM}$.

The **forwardCall**(\cdot) function respects the registers calling convention of Assumption 2.

Moreover, it only insists that returnbacks are done to the **forwardReturn**(\cdot) entry point.

For well-behaving contexts, that entry point just acts as a proxy for the real return entry point, so no semantics change is introduced there.

Moreover, **forwardReturn**(\cdot) also respects Assumption 2.

Well-behaving contexts will not call any entry point in $\llbracket Sys \rrbracket_{AIL}^{JEM}$ except for **testObject**(\cdot) and **registerObject**(\cdot).

The former is a correct implementation of **instanceof**, as already discussed.

The second one will never abort for well-behaving contexts since only new objects will be registered.

Therefore, the additional functionalities of $\llbracket Sys \rrbracket_{AIL}^{JEM}$ do not alter the semantics of correct programs.

Because of this, $\llbracket \cdot \rrbracket_{AIL}^{JEM}$ satisfies Assumption 6.

So we have: $\forall \mathcal{C}_1, \mathcal{C}_2. \mathcal{C}_1 \simeq_{ctx} \mathcal{C}_2 \iff \llbracket \mathcal{C}_1 \rrbracket_{AIL}^{JEM} \stackrel{w-b}{\simeq}_{ctx}^{AIL} \llbracket \mathcal{C}_2 \rrbracket_{AIL}^{JEM}$.

JEM-like-behaving contexts in AIL are a subset of all the AIL contexts, we have $\llbracket \mathcal{C}_1 \rrbracket_{AIL}^{JEM} \simeq_{ctx} \llbracket \mathcal{C}_2 \rrbracket_{AIL}^{JEM} \Rightarrow \llbracket \mathcal{C}_1 \rrbracket_{AIL}^{JEM} \stackrel{w-b}{\simeq}_{ctx}^{AIL} \llbracket \mathcal{C}_2 \rrbracket_{AIL}^{JEM}$.

Because we changed the calling convention, we need to show that all of these well-behaved contexts, which used the (\cdot) calling convention, can be made to adopt the $\llbracket \cdot \rrbracket_{AIL}^{JEM}$ one.

This can be done by adding the following instructions before a jump to a module – intuitively at the single exit point of those contexts:

- assume the address where to jump is stored in $\mathbf{r}_n, \mathbf{r}_i$ where n and i are any register with index from 0 to 4 (it can't be other indices as they are needed by the $\langle \cdot \rangle$ calling convention)
- store \mathbf{r}_n into \mathbf{r}_4 and \mathbf{r}_i into \mathbf{r}_3
- load 1 into \mathbf{r}_i and $2*\mathcal{N}_w$ into \mathbf{r}_n
- **jmp** $\mathbf{r}_n \mathbf{r}_i$

We can therefore chain the implications above to conclude the thesis. \square

Lemma 2 (Method emulation is correct). $\text{emulate}(a, M) = m : M_t \iff \langle m \rangle = a$.

Proof Sketch. This can be seen by Rules **Emulate - addresses** and **Emulate - addresses - fail** and the constitution of M_0 in Rule **methodKnowledge**. \square

Lemma 3 (Value emulation is correct). $\text{emulate}(w : t, V) = E, V' \iff \langle E \rangle = w$.

Proof Sketch. This can be seen by Rules **Emulate - values - unit** to **Emulate - values - external old** and **Emulate - values - external new** to **Emulate - values - intern - fail** and by the constitution of V_0 in Rule **objectsKnowledge** and by the update on V in all **emulate(·)** rules in Appendix C.4. \square

Proof of Lemma 1.

As a helper, here is a table of all additionally-defined functions and whether they can abort (i.e., call **halt**) or not.

testObj	yes
registerObj	yes
forwardCall	yes, twice
forwardReturn	yes, twice
classOf	no
isinternal	no
updatemaskingtable	no (but it calls registerObj)
loadobject	no (but it calls lookup in \mathcal{T})
lookup in \mathcal{T}	yes
maskingTable	no
dynamictypechecks	yes, twice plus it calls testObj
maskObjectid	no
signatureof	no
loadData	no
storedata	no
extraCode	no
extraData	no
append	no
replace	no
method ep	yes
return ep	yes

Proof Sketch. This proof is split in 2 cases, one for each direction of the co-implication.

By Assumption 3 we know that \checkmark is only produced by triggered checks.

$\forall \mathcal{C}, \llbracket \mathcal{C} \rrbracket = (m_c, m_d, t). \overline{\alpha} \checkmark \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C} \rrbracket_{\text{AIL}}^{\text{JEM}}) \Rightarrow \text{emulate}(\overline{\alpha}, t) = \text{fail}$. By inspecting the pseudocode of entry points created by `protect(·)`, we can see what function aborted, causing a \checkmark in the trace:

method entry point :

- 0m it can abort if the jump does not come from $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$ with return address being the forward return entry point
- 1m `dynamicTypechecks()` can abort on ill-typed or unregistered parameters;
- 2m `loadObjects()` can abort on fake or non-existing parameters (actually, it's lookup in \mathcal{T} that aborts);

forwardReturn entry point :

- 1r it can abort if a return is made when no outcall was done (empty \mathcal{S});
- 2r it can abort if the return is not made by the module that was called (mismatching \mathbf{r}_0);

return entry point :

- 3r `dynamicTypechecks()` can abort on an ill-typed return value;
- 4r it can abort if the module jumping here is not $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$.

violation of the PMA access control policy :

- 5r a return to a previously-communicated address that is not executable

We can then analyse the action preceding the \checkmark .

The proof proceeds by induction on the size of $\overline{\alpha}$.

$\overline{\alpha} = \emptyset$ Trivial.

$\overline{\alpha} = \alpha_1, \dots, \alpha_n$ where $n \% 2 = 1$ so $\alpha_n = \gamma_n?$. The proof proceeds by case analysis on $\gamma_n?$.

call $a \ w_0, \dots, w_m?$:

Let us consider first the case when a is a method entry point, considering the case of a successful forward by $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$.

w_0 is the caller id placed by PMA, if it is not $\llbracket \text{Sys} \rrbracket_{\text{AIL}}^{\text{JEM}}$, the execution aborts as in point 0m above, so Rule **Emulate - call -fail2** fails.

w_1 - w_5 are unused registers.

w_6 is the current object, if that is not an encoding of an object that implements class t (Point 2m above), Rule **Emulate - values - intern - fail** fails.

w_i with $i \in 7..l$, if that is not a valid encoding of a value of type t_i (Points 1m and 2m above), then Rules **Emulate - values - unit fail**, **Emulate - values- bool fail**, **Emulate - values - external old - fail**, **Emulate - values - intern - fail** and **Emulate - obj -ex fail fail**. These rules cover all possible cases of ill-typed parameters for **Unit** (Rule **Emulate - values - unit fail**), **Bool** (Rule **Emulate - values- bool fail**), **Obj** (Rule **Emulate - obj -ex fail**), or class type (Rules **Emulate - values - external old - fail** and **Emulate - values - intern - fail**).
 w_j with $l < j < m$, those registers are not used.

Let us consider when a is an entry point of $\llbracket Sys \rrbracket_{AIL}^{JEM}$; considerations for words that are not parameters (so w_1 to w_6 and 9 and beyond) are analogous to those above

- a is **testObj()**

w_7 can be a non-existing object id, in which case Rule **Emulate - call - testobj - w fails**.

There are no checks that the class encoding passed via w_8 is correct. When called from secure components, this (and the next) function are always called with correct class encodings. If the external code puts something else inside, it may try to cheat, but we'll find out because we test with correct class encodings. Otherwise, it could put anything inside, an object of a type that we don't know and don't use. In this case $\llbracket Sys \rrbracket_{AIL}^{JEM}$ would not be able to know this class. So we cannot filter based on known class encodings.

- a is **registerObj()**

w_7 can be an existing object id, in which case Rule **Emulate - call - regobj - w fails**.

- a is **forwardCall()**

w_0 can be the id of the module that last made a call (so no return was made), so Rule **Emulate - call - fwcall -w1 fails**.

w_3 can be the $\llbracket Sys \rrbracket_{AIL}^{JEM}$ module id, so someone is trying to forward a call to $\llbracket Sys \rrbracket_{AIL}^{JEM}$ via $\llbracket Sys \rrbracket_{AIL}^{JEM}$, so Rule **Emulate - call - fwcall -w2 fails**.

w_5 is the address to return to, if that (plus the id w_0) is not an executable address (Point 5r above), Rule **Emulate - call - fail fails**.

- a is **forwardReturn()**

If the stack \mathcal{S} is empty, someone is returning without any call being made, so Rule **Emulate - call - fwret - w2 fails**.

If w_0 is not the module id of the last module that made a call, Rule **Emulate - call - fwret - w1 fails**.

ret a $w, w'?$:

If a is not the address of **forwardReturn()**, Rule **Emulate - returnback - fail2 fails**.

So a let's consider a to be the address of `forwardReturn()`.
 If the action's index $h = 1$, then no returns can be made (Point 1r above) and Rule **Emulate - call - fwret - w2 fails**.
 Let the last called method have return type t and come from module with id id .
 If w is not a valid encoding of a value of type t , see previous case (Point 3r above).
 If w' is not id , Rule **Emulate - returnback - fail fails** (Point 2r above).

Having covered all cases when a \checkmark can be generated in traces, this case holds.

$\forall \mathcal{C}, \llbracket \mathcal{C} \rrbracket = (m_c, m_d, t). \bar{\alpha} \checkmark \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C} \rrbracket_{\text{AIL}}^{\text{JEM}}) \Leftarrow \text{emulate}(\bar{\alpha}, t) = \text{fail}$. The proof proceeds by analysing all cases that generate a *fail* (in Appendix C.4).

- Rule Emulate - returnback - fail** In this case the check inserted by `checkShortcutting()` ensures the \checkmark is generated.
- Rule Emulate - returnback - fail2** In this case the check inserted by `checkLocalStack()` ensures the \checkmark is generated.
- Rule Emulate - addresses - fail** The PMA architecture ensures the only addresses one can jump to is a valid entry point.
- Rule Emulate - values - unit fail** In this case the check inserted by `dynamicTypechecks()` ensures the \checkmark is generated.
- Rule Emulate - values- bool fail** In this case the check inserted by `dynamicTypechecks()` ensures the \checkmark is generated.
- Rule Emulate - values - external old - fail** In this case if the object id is a non-existent id, then `loadObjects()` will call `lookup` on \mathcal{T} that will abort, ensuring the \checkmark is generated. Otherwise, if the object id is ill-typed, `dynamicTypechecks()` calls `testObject()`, the check there ensures the \checkmark is generated.
- Rule Emulate - values - intern - fail** In this case if the object id is a non-existent id, then `loadObjects()` will call `lookup` on \mathcal{T} that will abort, ensuring the \checkmark is generated. Otherwise, if the object id is ill-typed, `dynamicTypechecks()` calls `testObject()`, the check there ensures the \checkmark is generated.
- Rule Emulate - obj -ex fail** In this case if the object id is a non-existent id, then `loadObjects()` will call `lookup` on \mathcal{T} that will abort, ensuring the \checkmark is generated. Otherwise, if the object id is ill-typed, `dynamicTypechecks()` calls `testObject()`, the check there ensures the \checkmark is generated.
- Rule intfor -symbol** Because we are only considered memories where no symbols are present, nor passed, this is never triggered.
- Rule Emulate - call - fail** In this case, the violation of the *PMA* access control policy ensures the \checkmark is generated.

- Rule Emulate - call - fail** In this case, the return jump to (r_0, r_5) will be halted by the *PMA*.
- Rule Emulate - call - fwcalls - w1** In this case the check by `forwardCall()` that the calling id is not the last id is triggered.
- Rule Emulate - call - fwcalls - w2** In this case the check by `forwardCall()` that the forwarding module id is not 1 is triggered.
- Rule Emulate - call - fwret - w1** In this case the check by `forwardReturn()` that the returning id is not the last caller's one is triggered.
- Rule Emulate - call - fwret - w2** In this case the check by `forwardReturn()` that the stack is empty is triggered.
- Rule Emulate - call - testobj - w** In this case the check by `testObj()` that the object to be registered is not fresh is triggered.
- Rule Emulate - call - regobj - w** In this case the check by `registerObj()` that the object to be tested exists is triggered.

Having considered all the cases when a *fail* is generated, this case holds. \square

Proof of Theorem 3.

Proof Sketch. We need to prove that:

`skeleton(C_1, C_2)` is correct and it produces well-typed code.

This can be verified by inspecting the `skeleton(\cdot)` definition in Appendix C.3.

`emulate($\bar{\alpha}, t$)` is correct, it produces well-typed code that correctly emulates all actions in $\bar{\alpha}$.

This proof proceeds by induction on the length of $\bar{\alpha}$. The base case in void, so only the inductive case is presented.

Assume $\bar{\alpha} = \alpha_1, \dots, \alpha_{k-1}, \alpha_k$. Denote $\alpha_1, \dots, \alpha_{k-1}$ with $\bar{\alpha}'$. The inductive hypothesis tells us that `emulate($\bar{\alpha}', t$)` is correct, it produces well-typed code that correctly emulates all actions in $\bar{\alpha}'$.

We need to prove that the emulation of α_k is correct. Formally, this is captured by: `emulate($\alpha_k, M_k, V_k + V_{k-1}, \overline{id_k}, \overline{m_k}, \overline{t_k}, i_k$)` = $A_k, V'_k, \overline{id_k}, i_k, \overline{m_k}, \overline{t_k}$.

The proof proceeds by case analysis on α_k .

call $a\ w_1, \dots, v_n?$ Rule **Emulate - call** ensures that the call is emulated by a well-typed method call to the correct method. The correctness of the method is given by Lemma 2 while the correctness of the arguments and the current object is given by Lemma 3.

call $a\ w_1, \dots, v_n!$ Rule **Emulate - callback** ensures that the state of oc is updated correctly.

ret $a\ w, w'?$ Rule **Emulate - returnback** ensures that this is emulated by returning the right value. The correctness of returned value is given by Lemma 3.

ret $a\ w, w'!$ Rule **Emulate - return** ensures that the state of `oc` is updated correctly.

✓ in this case the code is empty, as produced by Rule **Emulate- fail**. This is because the preceding trace must have generated a *fail*, as guaranteed by Lemma 1.

$\text{diff}(\alpha_1, \alpha_2, i, M, V, \overline{m}, \overline{t})$ is correct, it produces well-typed code and it considers all possible cases where α_1 and α_2 can be different.

This can be verified by inspecting the $\text{diff}(\cdot)$ definition in Appendix C.5.

Possible cases of different actions:

Different action :

call-return Rule **Diff-acts**
call-tick Rule **Diff-acts2**
return-tick Rule **Diff-acts3**
call-nothing Rule **Diff-length-call**
return-nothing Rule **Diff-length-return**
tick-nothing Rule **Diff-length-tick**

Same action, different parameter :

outcall : **call $a\ w_1, \dots, w_n!$**
different address a Rule **Diff-calls-methods**
different word w_0 This is always 1
different word w_1 to w_4 Rule **Diff-r0-r4**
different word w_5 This is always $3 * \mathcal{N}_w$
different word w_6 Rule **Diff-callee**
different word w_i with $i \in 7..n$ Rules **Diff-calls2-parameters - primitive**
to **Diff-calls2-parameters - external**
return : **ret $a\ w, w'!$**
different address a Rule **Diff-rets-addr**
different id id This is always 1
different word w Rules **Diff-rets-primitive**, **Diff-rets-internal** and **Diff-rets-external**

\mathcal{C}_1 and \mathcal{C}_2 have the same **import** requirements, otherwise it's trivial to differentiate them. Rule **Emulate** and the definition of **skeleton**(\cdot) ensure that all required interfaces are implemented in \mathcal{C} as well as all objects, so this points holds.

□

Proof of Theorem 2.

$\mathcal{C} \frown \mathcal{C}_1$ and $\mathcal{C} \frown \mathcal{C}_2$. *Proof.* Trace semantics deals with *sets* of traces, while the algorithm inputs *single* traces. Moreover, these single traces must be the same up to a !-decorated action. The two different single traces are obtained as follows. Since $\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \not\sqsubseteq \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}$, we have that $\text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}) \neq \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}})$. So there exists a trace $\overline{\alpha}$ that belongs to either only $\text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}})$ or only to $\text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}})$ but not to both. Assume wlog that $\overline{\alpha} \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}})$. The trace $\overline{\alpha}$ can be split in two parts $\overline{\alpha}_s$ and α_d such that $\overline{\alpha} = \overline{\alpha}_s \alpha_d$ and such that $\overline{\alpha}_s$ is the longest prefix of all traces of $\overline{\alpha}_2$. Let i be the index of α_d in $\overline{\alpha}$. So, there exists a trace $\overline{\alpha}' \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}})$ that can be split in two parts $\overline{\alpha}_s$ and α'_d such that $\overline{\alpha}' = \overline{\alpha}_s \alpha'_d$ and $\alpha_d \neq \alpha'_d$. Trace $\overline{\alpha}'$ always exists, it could be an empty trace, it could be composed by an empty $\overline{\alpha}_s$ and, possibly, by an empty α'_d .

Theorem 3 tells us that we can use $\langle\langle \cdot \rangle\rangle$ to get the component that differentiates between \mathcal{C}_1 and \mathcal{C}_2 . \square

Define $\mathcal{C}[\mathcal{C}_1] \uparrow \not\Leftarrow \mathcal{C}[\mathcal{C}_2] \uparrow$ as $\mathcal{C}[\mathcal{C}_1] \uparrow \wedge \mathcal{C}[\mathcal{C}_2] \downarrow$ or $\mathcal{C}[\mathcal{C}_2] \uparrow \wedge \mathcal{C}[\mathcal{C}_1] \downarrow$.

Proof of Theorem 4.

Proof. The equivalence is split into two subgoals (directions). Apply Theorem 1 to fulfill direction \Leftarrow . The direction \Rightarrow is restated using Assumption 5 into: $\forall \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_1 \simeq_{\text{ctx}} \mathcal{C}_2 \Rightarrow \llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \sqsubseteq \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}$. The contrapositive of this direction is: $\forall \mathcal{C}_1, \mathcal{C}_2, \llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \not\sqsubseteq \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}} \Rightarrow \mathcal{C}_1 \not\simeq_{\text{ctx}} \mathcal{C}_2$. Apply Theorem 2 to prove this direction. \square

Corollary 2 (Modular full-abstraction for compiled components). $\forall \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4$. $\mathcal{C}_1; \mathcal{C}_2 \simeq_{\text{ctx}} \mathcal{C}_3; \mathcal{C}_4 \iff \text{link}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}, \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}) \simeq_{\text{ctx}} \text{link}(\llbracket \mathcal{C}_3 \rrbracket_{\text{AIL}}^{\text{JEM}}, \llbracket \mathcal{C}_4 \rrbracket_{\text{AIL}}^{\text{JEM}})$.

Proof. Corollary 1 follows from Theorem 4 and modularity of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$.

Let $\mathcal{C}_1 = C_1^1, \dots, C_1^n, \mathcal{C}_2 = C_2^1, \dots, C_2^m, \mathcal{C}_3 = C_3^1, \dots, C_3^l, \mathcal{C}_4 = C_4^1, \dots, C_4^k$.

By Definition 1 we have that $n + m = l + k$.

Let $\mathcal{C} = C_1^1, \dots, C_1^n, C_2^1, \dots, C_2^m$ and $\mathcal{C}' = C_3^1, \dots, C_3^l, C_4^1, \dots, C_4^k$.

Theorem 4 yields that, $\mathcal{C} \simeq_{\text{ctx}} \mathcal{C}' \iff \llbracket \mathcal{C} \rrbracket_{\text{AIL}}^{\text{JEM}} \simeq_{\text{ctx}} \llbracket \mathcal{C}' \rrbracket_{\text{AIL}}^{\text{JEM}}$.

By definition of $\llbracket \cdot \rrbracket_{\text{AIL}}^{\text{JEM}}$, we have $\text{link}(\llbracket C_1^1 \rrbracket_{\text{AIL}}^{\text{JEM}}, \dots, \llbracket C_1^n \rrbracket_{\text{AIL}}^{\text{JEM}}, \llbracket C_2^1 \rrbracket_{\text{AIL}}^{\text{JEM}}, \dots, \llbracket C_2^m \rrbracket_{\text{AIL}}^{\text{JEM}}) \simeq_{\text{ctx}} \text{link}(\llbracket C_3^1 \rrbracket_{\text{AIL}}^{\text{JEM}}, \dots, \llbracket C_3^l \rrbracket_{\text{AIL}}^{\text{JEM}}, \llbracket C_4^1 \rrbracket_{\text{AIL}}^{\text{JEM}}, \dots, \llbracket C_4^k \rrbracket_{\text{AIL}}^{\text{JEM}})$, so the thesis holds. \square

Theorem 5 (Contextual equivalence is congruent with linking). $\forall P_1, P_2, P_3$. $P_2 \simeq_{\text{ctx}} P_3, \text{link}(P_1, P_2) \simeq_{\text{ctx}} \text{link}(P_1, P_3)$.

Proof Sketch. $P_2 \simeq_{\text{ctx}} P_3$ tells us that P_2 and P_3 have the same modules.

If there is any overlapping module (except $\llbracket Sys \rrbracket_{\text{AIL}}^{\text{JEM}}$) between P_1 and P_2 or P_3 , the result will be an empty memory.

If there is no overlap, then linking simply merges the modules, which trivially preserves \simeq_{ctx} . \square

Proof of Corollary 1.

Proof. By Corollary 2, we have that $\text{link}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}, \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}) \simeq_{\text{ctx}} \text{link}(\llbracket \mathcal{C}_3 \rrbracket_{\text{AIL}}^{\text{JEM}}, \llbracket \mathcal{C}_4 \rrbracket_{\text{AIL}}^{\text{JEM}})$.

By Theorem 5 we have that $\text{link}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}, \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}) \simeq_{\text{ctx}} \text{link}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}, P)$.

By Theorem 5 we have that $\text{link}(\llbracket \mathcal{C}_3 \rrbracket_{\text{AIL}}^{\text{JEM}}, \llbracket \mathcal{C}_4 \rrbracket_{\text{AIL}}^{\text{JEM}}) \simeq_{\text{ctx}} \text{link}(\llbracket \mathcal{C}_3 \rrbracket_{\text{AIL}}^{\text{JEM}}, P')$.

By transitivity of contextual equivalence, the thesis holds. \square

Proof of Theorem 2

Proof. $\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}} \not\equiv \llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}$ means that $\text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}) \neq \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}})$, so there exists a trace $\bar{\alpha}\alpha_1 \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}})$ but $\bar{\alpha}\alpha_1 \notin \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}})$.

Let $\bar{\alpha}$ be the longest odd-length prefix of traces from $\text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}})$ and $\text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}})$ such that $\bar{\alpha}\alpha_1 \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}})$ and $\bar{\alpha}\alpha_2 \in \text{Traces}_{\text{AIL}}(\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}})$ and $\alpha_1 \neq \alpha_2$. The index of α_1 and α_2 is even, so they are generated by $\llbracket \mathcal{C}_1 \rrbracket_{\text{AIL}}^{\text{JEM}}$ and $\llbracket \mathcal{C}_2 \rrbracket_{\text{AIL}}^{\text{JEM}}$ respectively. Theorem 3 tells us that $\langle\langle \mathcal{C}_1, \mathcal{C}_2, \bar{\alpha}\alpha_1, \bar{\alpha}\alpha_2 \rangle\rangle$ yields the context that differentiates between \mathcal{C}_1 and \mathcal{C}_2 . \square

References

- [1] Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, London, UK, 1999.
- [2] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83.
- [3] Martín Abadi, Jérémy Planul, and Gordon Plotkin. Layout randomization and nondeterminism. *Electr. Notes Theo. Comp. Sci.*, 298:29–50, 2013.
- [4] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM TISSEC*, 15(2):8:1–8:29, July 2012.
- [5] Amal Ahmed. Verified Compilers for a Multi-Language World. In *SNAPL 2015*, volume 32 of (*LIPICs*), pages 15–31, 2015.
- [6] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. *SIGPLAN Not.*, 43(9):157–168, September 2008.
- [7] Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not.*, 46(9):431–444, 2011.
- [8] Ittai Anati, Shay Gueron, S Johnson, and V Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *HASP’13*, volume 13, 2013.
- [9] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hrițcu, David Pichardie, Benjamin C. Pierce, Randy Pollock, and Andrew Tolmach. A verified information-flow architecture. *SIGPLAN Not.*, 49(1):165–178.
- [10] Ioannis G. Baltopoulos and Andrew D. Gordon. Secure compilation of a multi-tier web language. In *TLDI ’09*, pages 27–38. ACM, 2009.
- [11] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM TISSEC*, 2010.
- [12] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not.*, 44(9):97–108, August 2009.
- [13] William J. Bowman and Amal Ahmed. Noninterference for free. In *ICFP ’15*, New York, NY, USA, 2015. ACM.
- [14] Michele Bugliesi and Marco Giunti. Secure implementations of typed channel abstractions. In *POPL ’07*, pages 251–262, 2007.
- [15] Ricardo Corin, Pierre-Malo Deniérou, Cédric Fournet, Karthikeyan Bhargavan, and James Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5):573–636, 2008.

- [16] Victor Costan and Srinivas Devadas. Intel SGX Explained.
- [17] Frank S. de Boer, Marcello M. Bonsangue, Martin Steffen, and Erika Ábrahám. A fully abstract semantics for UML components. In *FMCO'04*.
- [18] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *POPL 2016*, pages 164–177, 2016.
- [19] D. Dolev and A. C. Yao. On the security of public key protocols. In *SFCS*, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society.
- [20] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. *SIGPLAN Not.*, 48(1):371–384, January 2013.
- [21] Daniele Gorla and Uwe Nestman. Full abstraction for expressiveness: History, myths and facts. *Math Struct Comp Science*, 2014.
- [22] Mike Hamburg, Paul Kocher, and Mark Marson. Analysis of Intel’s Ivy Bridge digital random number generator. *Cryptography Research, Inc.*, 2012.
- [23] Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. *SIGPLAN Not.*, 46(1):133–146, January 2011.
- [24] Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. Local memory via layout randomization. In *CSF '11*, pages 161–174. IEEE, 2011.
- [25] Alan Jeffrey and Julian Rathke. Java Jr.: fully abstract trace semantics for a core Java language. In *ESOP'05*, pages 423–438, 2005.
- [26] Yanis Juglaret and Catalin Hritcu. Secure compilation using micro-policies (Extended Abstract). In *FCS 2015*, 2015.
- [27] Yannis Juglaret, Cătălin Hrițcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *29th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, July 2016. To appear.
- [28] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.
- [29] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *SP '10*, pages 143–158. IEEE, 2010.
- [30] Frank McKeen et. al. Innovative instructions and software model for isolated execution. In *HASP '13*, 2013.

- [31] Job Noorman et. al. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *USENIX'13'*.
- [32] Joachim Parrow. General conditions for full abstraction. *Math. Struc. Comp. Sci.*
- [33] Joachim Parrow. Expressiveness of process algebras. *Electronic Notes in Theoretical Computer Science*, 209(0):173 – 186, 2008.
- [34] Marco Patrignani. *The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures*. PhD thesis, KU Leuven, Leuven, Belgium, May 2015.
- [35] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, April 2015.
- [36] Marco Patrignani and Dave Clarke. Fully abstract trace semantics for protected module architectures. *Elsevier COMLAN*, 42(0):22 – 45, 2015.
- [37] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
- [38] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, April 2006.
- [39] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS 2012*.
- [40] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *Principles of Programming Languages*, pages 161–172. ACM, 2004.
- [41] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. *SIGPLAN Not.*, 49(1):425–437, January 2014.
- [42] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX*, pages 29–46, 2010.
- [43] Jonathan Woodruff et. al. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA '14*, pages 457–468, 2014.